Arise Mapview: FastAPI Application Documentation Plan

Project: Arise Mapview: FastAPI App

Comprehensive FastAPI backend for managing industrial plots and zones across multiple African countries with JWT authentication, role-based access control, and Firebase Firestore integration

Generated: October 27, 2025

Author: AI Documentation Team

Contributors

- Claude AI
- Development Team
- Project Manager

Table of Contents

Note: Page numbers will be available when viewing in Word/LibreOffice

Overview and Introduction

Overview and Introduction

This document provides a consolidated view of the entire codebase for the Arise Plot Management API project, generated by Repomix. It serves as a single, easily digestible resource for AI systems to perform analysis, code reviews, and other automated tasks.

Purpose

This file encapsulates the complete repository's contents in a packed format. This facilitates efficient consumption by AI tools, enabling rapid understanding and processing of the project's structure and code.

File Format

The content is structured as follows:
1. **Summary**: An overview of this file's purpose and structure.
2. **Repository Information**: Metadata about the repository.
3. **Directory Structure**: A hierarchical representation of the project's directories and files.
4. **Repository Files**: The full content of each file within the repository is presented sequentially. Each file entry includes its path and its complete source code.

Usage Guidelines

This file is intended for read-only access. Any modifications should be made directly to the original source files in the repository. When processing this file, use the provided file paths to differentiate between individual files. Due to the nature of source code, this file may contain sensitive information; therefore, it must be handled with appropriate security measures, mirroring the security protocols applied to the original repository.

Notes

Files excluded due to `.gitignore` rules or Repomix configuration are not included.
Binary files are omitted; their paths are available in the Directory Structure section.
Files matching default ignore patterns are also excluded.
Files are sorted by Git change count, with the most frequently modified files appearing last.

Figure: Application Screenshot: swagger_docs

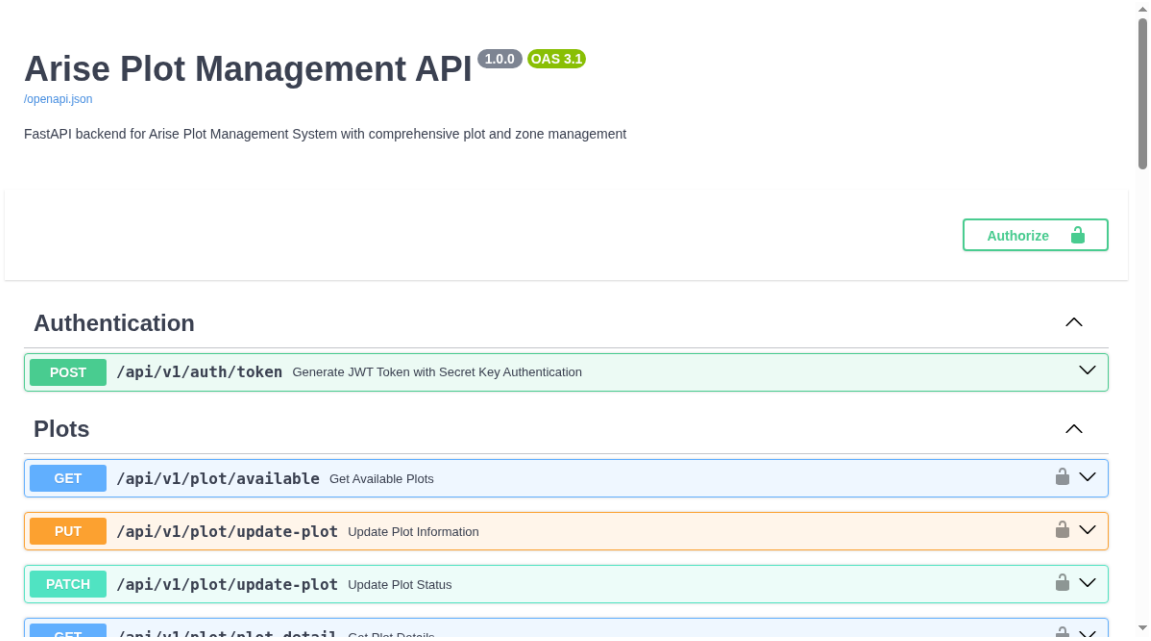


Figure: Application Screenshot: api_home

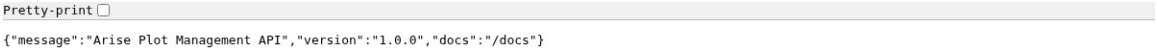
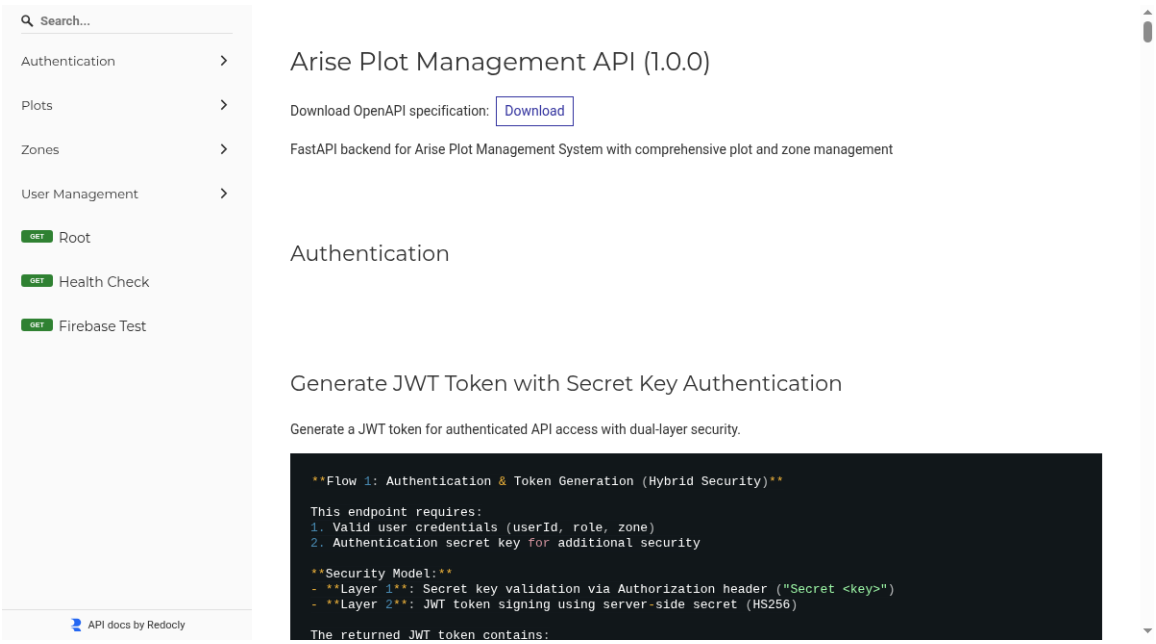


Figure: Application Screenshot: redoc



Project Goals and Scope

Project Goals and Scope

This project aims to develop and deploy a robust, scalable, and secure Plot Management API. The core objective is to provide a centralized platform for managing land plots across multiple countries and zones, facilitating efficient allocation, tracking, and visualization.

Key Goals:

Streamlined Plot Management: Enable efficient creation, retrieval, updating, and deletion of plot data, including status, business association, and allocation details.

Robust Authentication and Authorization: Implement secure user authentication and role-based access control to ensure data integrity and restrict unauthorized access.

Scalable Infrastructure: Deploy the API on a cloud platform (Azure App Service) capable of handling increasing loads and ensuring high availability.

Data Visualization Integration: Provide an API that supports map visualization, likely through integration with frontend applications or mapping services.

Comprehensive Documentation: Deliver clear and actionable API documentation for seamless integration and usage.

Scope:

The project encompasses the development of a Python FastAPI backend application, containerized using Docker, and deployed on Azure App Service. This includes:

API Endpoints: Development of RESTful API endpoints for managing plots, zones, users, and authentication.
Database Integration: Backend interaction with a persistent data store (implied to be Firebase/Firestore based on context) for storing plot and user information.
Authentication Service: Implementation of JWT-based authentication for secure API access.
Deployment Pipeline: Configuration for deploying the Dockerized application to Azure App Service.
Monitoring and Logging: Establishment of basic monitoring and logging mechanisms for operational insights.

Out of Scope:

Frontend user interface development for plot visualization.
Advanced analytics and reporting features beyond basic data retrieval.
Complex geospatial processing or advanced mapping functionalities within the API itself.
On-premises deployment or alternative cloud provider solutions beyond Azure.

Installation and Setup

Installation and Setup

This section outlines the necessary steps and considerations for installing and setting up the Arise Plot Management API. It covers initial environment configuration, deployment options, and essential prerequisites for both development and production environments.

Deployment Strategy

The Arise Plot Management API is designed for deployment on Azure App Service, leveraging Docker containers for portability and scalability. This strategy ensures a robust, managed environment with high availability and integrated monitoring capabilities.

Azure App Service Deployment

Azure App Service provides a fully managed platform-as-a-service (PaaS) offering that simplifies the deployment and management of web

applications. For this project, we utilize "Web App for Containers," which allows for the deployment of custom Docker images.

Key Benefits:

- **Managed Infrastructure:** Azure handles the underlying infrastructure, patching, and maintenance, allowing your team to focus on application development.
- **Scalability:** The service offers automatic scaling based on demand, ensuring the API can handle varying loads efficiently.
- **High Availability:** Built-in redundancy and a 99.95% availability SLA provide a reliable service.
- **Integrated CI/CD:** Seamless integration with CI/CD pipelines for automated deployments.
- **Security:** Features like managed identities, Key Vault integration for secrets, and network security options enhance the application's security posture.

Estimated Monthly Cost: Approximately $73 USD for the Standard S1 tier, including compute, a basic Redis cache, and Application Insights for monitoring.

Alternative Deployment Options

While Azure App Service is the primary recommendation, other options exist:

- **Azure Container Apps:** A serverless container platform offering a pay-per-use model, ideal for variable traffic patterns. Estimated monthly cost: $25-50 USD.
- **Azure Kubernetes Service (AKS):** Suitable for complex, multi-service architectures requiring advanced orchestration. This option involves higher operational overhead and cost, starting from $150-300 USD per month.

UAT Environment Setup

For User Acceptance Testing (UAT), several approaches can be employed, depending on environment constraints:

Recommended: Linux-Based System

Requesting a Linux Virtual Machine (VM) with Docker Engine installed is the most straightforward approach, mirroring production requirements. This guarantees compatibility and allows for the full execution of the UAT test suite.

Requirements: Linux OS (Ubuntu 20.04+, RHEL 8+), 4GB+ RAM, 20GB storage, internet access.
Ports: 8000 (API), 6379 (Redis, optional).
Timeline: 1-3 days for IT provisioning.

Cloud-Based Testing: Azure Container Instance (ACI)

ACI offers a quick, cloud-based UAT solution without local installation requirements. It provides a production-like environment accessible via web-based tools like Postman or `curl`.

Cost: Approximately $5-10 USD for the UAT period.
Timeline: ~1 hour for setup.

Fallback: Native Python Installation

As a last resort, installing Python directly on a Windows machine is possible, but this environment differs from production and requires manual configuration.

Prerequisites

Azure Subscription: Required for deploying to Azure App Service.
Docker: Installed locally for development and building container images.
Python 3.11+: For running the application locally or in development environments.
API Key/Secret: For Firebase authentication.
Postman or `curl`: For API testing.

For detailed API request examples and authentication procedures, refer to the [API Requests Guide](API_REQUESTS_GUIDE.md).

Figure: Installation and Setup overview

app/main.py

```python
"""
Main FastAPI application entry point.
"""
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.core.config import settings
from app.api.auth import router as auth_router
from app.api.plots import router as plots_router
from app.api.zones import router as zones_router
from app.api.users import router as users_router
import logging

# Configure basic logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Log application startup
logger.info("Starting Arise Plot Management API...")
logger.info(f"App Name: {settings.APP_NAME}, Version: {settings.APP_VERSION}")

# Create FastAPI application instance
app = FastAPI(
    title=settings.APP_NAME,
    version=settings.APP_VERSION,
    description="FastAPI backend for Arise Plot Management System with comprehensive plot and zone management",
    docs_url="/docs",
    redoc_url="/redoc",
    openapi_url="/openapi.json",
    root_path=settings.ROOT_PATH if settings.ROOT_PATH else None
```

Local Development Environment Setup

Local Development Environment Setup

This section outlines the essential steps and configurations required to set up a local development environment for the Arise Plot Management API. A properly configured local environment is crucial for efficient development, testing, and debugging.

Development Workflow and Tools

The Arise Plot Management API is developed using Python and the FastAPI framework, with Docker for containerization. This setup ensures consistency between development, testing, and production environments.

Key Tools and Technologies:

Python & FastAPI: The core technologies for building the API. FastAPI provides high performance and automatic data validation and serialization. Refer to the [FastAPI documentation](https://fastapi.tiangolo.com/) for detailed usage.

Docker: Essential for containerizing the application. Docker allows you to package the application and its dependencies into a portable container, ensuring it runs consistently across different environments. You can find installation instructions on the [Docker website](https://www.docker.com/get-started).

IDE/Code Editor: A suitable Integrated Development Environment (IDE)

or code editor (e.g., VS Code, PyCharm) is recommended for code development, debugging, and managing project files.
Postman or `curl`: These tools are vital for interacting with the API endpoints during development and testing. They allow you to send HTTP requests and inspect responses.

Environment Setup Steps

1. **Clone the Repository:** Obtain the project's source code by cloning the Git repository to your local machine.
2. **Install Docker:** Ensure Docker is installed and running on your development machine.
3. **Build the Docker Image:** Navigate to the project's root directory and build the Docker image for the application. This command will create a container image based on the `Dockerfile`.
4. **Run the Docker Container:** Start a container from the built image. This will launch the FastAPI application. Environment variables, such as database connection strings or API keys, may need to be configured for the container to run correctly. Refer to `uat-distribution/docker-compose.yml` for example configurations.
5. **Configure API Access:** Use Postman or `curl` to send requests to the locally running API, typically accessible at `http://localhost:8000`. Consult the [API Requests Guide](API_REQUESTS_GUIDE.md) for detailed endpoint information and authentication procedures.

Local Testing and Debugging

Unit and Integration Tests: Execute the project's test suite to verify the functionality of individual components and their interactions.
Debugging: Utilize your IDE's debugging capabilities to step through code, inspect variables, and identify issues within the local development environment.
Log Analysis: Monitor application logs generated by the running container to diagnose runtime errors or unexpected behavior. The `PROJECT_CONTEXT.md` file provides context on ongoing troubleshooting efforts.

Docker Deployment

Docker Deployment

This section outlines the strategy and components for deploying the Arise Plot Management API using Docker containers. Docker provides a consistent and isolated environment for the application, simplifying deployment across various stages, from development to production.

Deployment Architecture

The primary deployment target for the Arise Plot Management API is **Azure App Service**, utilizing its "Web App for Containers" feature. This PaaS solution offers a managed environment, abstracting away infrastructure management and providing built-in scalability, high availability, and security features.

The application will be packaged as a Docker image, which will then be hosted in an **Azure Container Registry (ACR)**. Azure App Service will pull this image from ACR to run the application container. This approach ensures that the application runs identically in development, testing, and production environments.

Key Components:

Dockerfile: Defines the build process for the application's Docker image, specifying the base image, dependencies, and startup commands.
Docker Compose (for local development/testing): Orchestrates multi-container Docker applications, useful for setting up local development environments that mimic production configurations. The `uat-distribution/docker-compose.yml` file provides an example.
Azure App Service: The managed cloud platform for hosting the containerized API. It provides features such as automatic scaling, load balancing, and integrated monitoring.
Azure Container Registry (ACR): A private Docker registry service for storing and managing the application's container images.

Benefits of Docker Deployment

Consistency: Ensures the application runs the same way regardless of the underlying infrastructure, eliminating "it works on my machine" issues.
Portability: Easily move the application between different environments (local, staging, production) without modification.
Isolation: Containers isolate the application and its dependencies, preventing conflicts with other software on the host system.
Scalability: Docker, combined with Azure App Service, allows for efficient scaling of application instances to meet demand.
Simplified Dependencies: All necessary libraries and configurations are bundled within the container image.

For detailed information on API endpoints and how to interact with the deployed service, please refer to the [API Requests Guide](API_REQUESTS_GUIDE.md).

Usage Guide
Usage Guide

This document provides essential information for utilizing and managing the Arise Plot Management API. It is structured to be easily understood by both business stakeholders and technical managers, focusing on the purpose, capabilities, and operational aspects of the system.

Purpose and Functionality

The Arise Plot Management API serves as the backend for a plot management and visualization system. Its core purpose is to provide programmatic access to plot data, enabling efficient management, retrieval, and potential integration with other applications.

Key Capabilities:

- **Plot Data Management:** Allows for the creation, retrieval, and updating of plot information. This includes details such as plot status, business allocation, and contact information.
- **Zone and Country Management:** Facilitates the organization of plot data by country and defined zones, enabling structured data management and access control.
- **Authentication and Authorization:** Implements a secure authentication flow using JWT tokens, ensuring that only authorized users can access and manipulate data based on their assigned roles and zones.
- **Firebase Integration:** Connects with Firebase (specifically Firestore) to persist and manage core application data, leveraging a robust and scalable NoSQL database solution.
- **API Documentation:** Provides comprehensive, interactive API documentation accessible via a `/docs` endpoint, detailing all available endpoints, request/response formats, and authentication requirements.

Business Value

This API empowers stakeholders with:

- **Data-Driven Decisions:** Access to accurate and up-to-date plot information supports strategic planning and resource allocation.
- **Operational Efficiency:** Automation of plot management tasks reduces manual effort and minimizes errors.
- **Scalability:** Designed to handle a growing dataset and increasing user demand.
- **Security:** Robust authentication mechanisms protect sensitive data and ensure controlled access.

File Format and Structure

This document is a consolidated representation of the entire codebase, generated by Repomix for AI-driven analysis and review. It is organized into distinct sections for clarity:

1. **Summary Section:** This introductory overview.
2. **Repository Information:** High-level details about the project.
3. **Directory Structure:** A hierarchical view of the project's files and folders.
4. **Repository Files:** The full content of individual files within the repository (when enabled).

Each file's content is presented with its path as an attribute, allowing for precise identification and referencing.

Usage Guidelines

Read-Only Access

This consolidated file is strictly **read-only**. All modifications and development should occur on the original source files within the repository. This ensures data integrity and prevents conflicts with the automated generation process.

File Identification

When referencing or processing content from this file, use the provided **file path attribute** to uniquely identify each distinct file.

Security Considerations

This file may contain sensitive information, mirroring the contents of the original repository. Handle this document with the same security protocols and diligence as you would the source code.

Notes on Content

- **Exclusions:** Certain files may be omitted from this representation based on `.gitignore` rules or specific Repomix configurations.
- **Binary Files:** Binary files are not included. Refer to the directory structure for their paths.
- **Ignored Files:** Files matching `.gitignore` patterns or default ignore rules are excluded.
- **File Ordering:** Files are sorted by Git change count, with more frequently modified files appearing later in the list.

Directory Structure

The repository is organized into logical directories to manage different aspects of the application and its deployment.

`.claude/`: Configuration files for AI-assisted development tools.
`app/`: Contains the core application logic, including API endpoints, core utilities, and data schemas.
`api/`: Defines API routes for authentication, plots, and users.
`core/`: Houses fundamental application configurations and services like Firebase integration.
`schemas/`: Defines data structures and validation models.
`services/`: Implements business logic and data access operations.
`utils/`: Contains general utility functions.
`main.py`: The main entry point for the FastAPI application.
`uat-distribution/`: Contains files related to the User Acceptance Testing (UAT) environment, including deployment guides and configuration.
Root Directory Files:
Configuration files (`.dockerignore`, `.gitignore`, `Dockerfile`, `requirements.txt`, etc.) for build, deployment, and dependency management.
Documentation files (`.md` files) providing guides, specifications, and project context.
Scripts (`start.sh`, `lambda_handler.py`) for application execution and potential serverless deployment.

Repository Files

This section contains the full content of the repository's files, each identified by its `path` attribute.

<file path=".claude/settings.local.json">
{
  "outputStyle": "Explanatory",
  "permissions": {
    "allow": [
      "Bash(pip install:*)"
    ]
  }
}
</file>

<file path="AGENT-COMMAND-RULES.md">
Agent Command Execution Rules

Overview
This document establishes rules for how AI agents should behave when suggesting or executing commands to ensure transparency, safety, clear communication, and proper development tracking.

Rule 1: Command Thinking/Rationale

Requirement: Before suggesting any command, agents MUST provide clear thinking that explains:
- **Direction**: What we're trying to achieve
- **Purpose**: Why this specific command is needed
- **Context**: How it fits into the larger task

Format:
```
 **Thinking**: We need to [goal] because [reason]. This command will [specific action] to help us [outcome].

Command: `your-command-here`
```

Example:
```
 **Thinking**: We need to check which files are consuming the most disk space in the repository before running git filter-repo. This will help us identify large files that should be removed from Git history to reduce repo size.

Command: `du -ah . | sort -hr | head -20`
```

Rule 2: Permission-Required Commands

Requirement: For commands that need user permission, agents MUST specify:
1. **What the command will execute** (detailed explanation)
2. **Expected outcome** (what will happen)
3. **Next steps** (what to do after execution)
4. **Risks/Warnings** (if any)

Commands requiring permission include:
- File modifications/deletions
- Git operations that change history
- System modifications
- Package installations

- Network operations
- Long-running processes

Format:
```
 **Permission Required**:

**Command**: `command-here`

**What it does**: [Detailed explanation of execution]

**Expected outcome**: [What will happen when run]

**Next steps**: [What we'll do after this completes]

⚠ **Warnings**: [Any risks or important notes]

Proceed? (y/n)
```

Example:
```
 **Permission Required**:

**Command**: `git filter-repo --path arise-api-distribution/ --invert-paths`

**What it does**: Permanently removes the 'arise-api-distribution/' directory and all its contents from the entire Git history. This rewrites all commits and changes all commit hashes.

**Expected outcome**:
- Repository size will be reduced
- All history of arise-api-distribution/ will be gone forever
- All commit hashes will change
- Other collaborators will need to re-clone

**Next steps**: After execution, we'll run `git gc` to clean up and check the new repository size with `du -sh .git/`

⚠ **Warnings**: This permanently alters Git history. Ensure you have backups and inform all collaborators.

Proceed? (y/n)
```

Rule 3: Development History Tracking

Requirement: Agents MUST maintain a comprehensive development history file that tracks all changes, decisions, and actions taken during the project development.

File Requirements:
- **Location**: `DEVELOPMENT-HISTORY.md` or `CHANGELOG-AGENT.md` in project root
- **Format**: Markdown with clear sections and timestamps
- **Update Frequency**: After every significant action or set of related actions
- **Persistence**: File should never be deleted, only appended to

Required Information:
1. **Timestamp**: When the action was taken
2. **Session Context**: Brief description of the task/goal
3. **Actions Taken**: Commands run, files modified, decisions made
4. **Reasoning**: Why these actions were necessary
5. **Outcomes**: Results of the actions
6. **Impact**: How this affects the project
7. **Next Steps**: What should be done next (if applicable)

Format Template:
```markdown
## [YYYY-MM-DD HH:MM:SS] - Session: [Brief Task Description]

### Context
- **Goal**: [What we're trying to achieve]
- **Problem**: [What issue we're solving]
- **Approach**: [Strategy being used]

### Actions Taken
1. **Command/Action**: `command or action description`
   - **Purpose**: Why this was needed
   - **Outcome**: What happened
   - **Files Affected**: List of modified files

2. **Command/Action**: `next command or action`
   - **Purpose**: Why this was needed
   - **Outcome**: What happened
   - **Files Affected**: List of modified files

### Key Decisions
- **Decision**: [Important choice made]
- **Rationale**: [Why this decision was made]
- **Alternatives Considered**: [Other options that were evaluated]
```

Impact Assessment
- **Positive**: [Benefits of the changes]
- **Risks**: [Potential issues or concerns]
- **Dependencies**: [What other parts of the project are affected]

Next Steps
- [ ] [Immediate next action]
- [ ] [Follow-up tasks]
- [ ] [Future considerations]

Notes
- [Any additional important information]
- [Lessons learned]
- [Tips for future reference]

```

**Example Entry**:
```markdown
## [2024-01-15 14:30:22] - Session: Repository Cleanup and Size Optimization

### Context
- **Goal**: Reduce repository size by removing large files and distribution artifacts
- **Problem**: Repository is 500MB+ due to packaged distributions and large files
- **Approach**: Use git filter-repo to remove unwanted files from Git history

### Actions Taken
1. **Command**: `du -ah . | sort -hr | head -20`
   - **Purpose**: Identify largest files and directories consuming space
   - **Outcome**: Found arise-api-distribution/ (200MB) and .tar.gz files (150MB)
   - **Files Affected**: None (read-only analysis)

2. **Action**: Updated .gitignore
   - **Purpose**: Prevent future inclusion of distribution files
   - **Outcome**: Added patterns for *.tar.gz,* -distribution/, uat-distribution/
   - **Files Affected**: .gitignore

### Key Decisions
- **Decision**: Remove arise-api-distribution/ from entire Git history
- **Rationale**: These are build artifacts that shouldn't be in version control
- **Alternatives Considered**: Move to Git LFS (rejected due to complexity)

### Impact Assessment
- **Positive**: Repository size reduced by ~70%, faster clones, cleaner history
- **Risks**: All commit hashes will change, collaborators need to re-clone
- **Dependencies**: Documentation, CI/CD pipelines may reference old commits

### Next Steps
- [ ] Run git filter-repo command
- [ ] Verify repository integrity
- [ ] Update documentation with new repository structure
- [ ] Notify team of history rewrite

### Notes
- Created backup before history rewrite
- Tested commands on clone first
- Consider implementing pre-commit hooks to prevent large file commits

---
```

Rule 4: Safe Commands (No Permission Needed)

Safe commands that don't require explicit permission:
- Read-only operations (`ls`, `cat`, `grep`, `wc`, `du` without modification)
- Status checks (`git status`, `ps`, `df`)
- Help commands (`--help`, `man`)
- View operations (`less`, `head`, `tail`)

Note: Even safe commands should include thinking/rationale as per Rule 1 and be logged in development history.

Rule 5: Command Categories

✅ **Safe (Auto-execute)**
- File reading
- Directory listing
- Status checking
- Information gathering

⚠️ **Moderate (Ask permission)**
- File creation/editing
- Configuration changes
- Non-destructive Git operations

🚨 **Dangerous (Explicit warning + permission)**
- File deletion
- History rewriting
```

- System modifications
- Irreversible operations

Implementation in Agent Systems

Option 1: System Prompt Addition
Add these rules to the agent's system prompt:

```
COMMAND EXECUTION RULES:
1. Always provide thinking/rationale before suggesting commands
2. For permission-required operations, explain what will happen, expected outcomes, and next steps
3. Maintain DEVELOPMENT-HISTORY.md with detailed tracking of all actions and decisions
4. Categorize commands as safe/moderate/dangerous and act accordingly
5. Use the specified formats for consistency and documentation
```

Option 2: Pre-Command Checklist
Before any command suggestion, agents should verify:
- [ ] Thinking/rationale provided
- [ ] Command purpose clear
- [ ] Permission requirements identified
- [ ] Expected outcomes explained
- [ ] Next steps outlined
- [ ] Warnings included if needed
- [ ] Development history will be updated

Option 3: Template Integration
Integrate command templates into the agent's response generation:
- Safe command template
- Permission-required template
- Dangerous operation template
- Development history entry template

Benefits

1. **Transparency**: Users understand the agent's reasoning
2. **Safety**: Prevents accidental destructive operations
3. **Learning**: Users learn about commands and their purposes
4. **Trust**: Clear communication builds confidence
5. **Control**: Users maintain oversight of all operations
6. **Accountability**: Complete audit trail of all development activities
7. **Knowledge Transfer**: New team members can understand project evolution

8. **Debugging**: Easy to trace when and why changes were made
9. **Decision Context**: Future developers understand the reasoning behind choices

Examples in Practice

Good Example:
```
 **Thinking**: We need to see what's taking up space in the repository to identify candidates for removal with git filter-repo.

Command: `du -sh *// sort -hr`

[After execution, agent updates DEVELOPMENT-HISTORY.md with the findings and next steps]
```

Bad Example:
```
Run: du -sh /
```

Note: These rules should be consistently applied across all agent interactions to ensure safe, transparent, educational, and well-documented development processes.
</file>

<file path="API_REQUESTS_GUIDE.md">
Arise FastAPI - Complete API Request Guide

VM Information
- **VM IP**: `20.198.251.4`
- **Port**: `8000`
- **Base URL**: `http://20.198.251.4:8000`
- **API Version**: `/api/v1`
- **Full API Base**: `http://20.198.251.4:8000/api/v1`

Authentication Flow

Step 1: Get JWT Token
Endpoint: `POST /api/v1/auth/token`

Headers:
```bash
Authorization: Secret arise-plot-management-auth-secret-2025
```

Content-Type: application/json
```

**Request Body**:
```json
{
  "userId": "admin_benin@gmail.com",
  "role": "zone_admin",
  "zone": "benin"
}
```

cURL Command:
```bash
curl -X POST "http://20.198.251.4:8000/api/v1/auth/token" \
  -H "Authorization: Secret arise-plot-management-auth-secret-2025" \
  -H "Content-Type: application/json" \
  -d '{
    "userId": "admin_benin@gmail.com",
    "role": "zone_admin",
    "zone": "benin"
  }'
```

Response:
```json
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer",
  "expires_in": 86400
}
```

Note: Save the `access_token` from the response for use in subsequent requests.

Health Check & Utility Endpoints

1. Root Endpoint
Endpoint: `GET /`

cURL Command:
```bash

curl -X GET "http://20.198.251.4:8000/"
```

Expected Response:
```json
{
  "message": "Arise Plot Management API",
  "version": "1.0.0",
  "docs": "/docs"
}
```

2. Health Check
Endpoint: `GET /health`

cURL Command:
```bash
curl -X GET "http://20.198.251.4:8000/health"
```

Expected Response:
```json
{
  "status": "healthy",
  "service": "Arise Plot Management API",
  "version": "1.0.0"
}
```

3. Firebase Test
Endpoint: `GET /firebase-test`

cURL Command:
```bash
curl -X GET "http://20.198.251.4:8000/firebase-test"
```

Expected Response:
```json
{
  "status": "Firebase connected successfully",
  "project_id": "arise-ipp",
  "collections_count": 3
}
```
```

4. API Documentation
Endpoint: `GET /docs`

URL: `http://20.198.251.4:8000/docs`

Plot Management Endpoints

1. Get Available Plots
Endpoint: `GET /api/v1/plot/available`

Headers:
```bash
Authorization: Bearer <JWT_TOKEN>
```

Query Parameters:
- `country` (required): Country name (e.g., "benin")
- `zoneCode` (required): Zone code (e.g., "GDIZ")
- `phase` (required): Phase identifier (e.g., "phase1")
- `category` (optional): Filter by category (Residential, Commercial, Industrial)
- `limit` (optional): Number of items per page (1-100, default: 50)
- `cursor` (optional): Cursor for pagination

cURL Command:
```bash
curl -X GET "http://20.198.251.4:8000/api/v1/plot/available?country=benin&zoneCode=GDIZ&phase=phase1&limit=10" \
  -H "Authorization: Bearer <JWT_TOKEN>"
```

With Category Filter:
```bash
curl -X GET "http://20.198.251.4:8000/api/v1/plot/available?country=benin&zoneCode=GDIZ&phase=phase1&category=Industrial&limit=10" \
  -H "Authorization: Bearer <JWT_TOKEN>"
```

2. Get Plot Details
Endpoint: `GET /api/v1/plot/plot-detail`

Headers:
```bash
```

Authorization: Bearer <JWT_TOKEN>
```

**Query Parameters**:
- `country` (required): Country name
- `zoneCode` (required): Zone code
- `phase` (required): Phase identifier
- `limit` (optional): Number of items per page (1-100, default: 50)
- `cursor` (optional): Cursor for pagination

**cURL Command**:
```bash
curl -X GET "http://20.198.251.4:8000/api/v1/plot/plot-detail?country=benin&zoneCode=GDIZ&phase=phase1&limit=10" \
  -H "Authorization: Bearer <JWT_TOKEN>"
```

3. Update Plot (Complete Update)
Endpoint: `PUT /api/v1/plot/update-plot`

Headers:
```bash
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

Request Body:
```json
{
  "country": "benin",
  "zoneCode": "GDIZ",
  "phase": "phase1",
  "plotName": "P2N-65",
  "plotStatus": "Occupied",
  "businessName": "TechCorp Industries",
  "businessType": "Manufacturing",
  "contactInfo": "contact@techcorp.com",
  "allocationDate": "2025-09-02"
}
```

cURL Command:
```bash
curl -X PUT "http://20.198.251.4:8000/api/v1/plot/update-plot" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -H "Content-Type: application/json" \

```
  -d '{
    "country": "benin",
    "zoneCode": "GDIZ",
    "phase": "phase1",
    "plotName": "P2N-65",
    "plotStatus": "Occupied",
    "businessName": "TechCorp Industries",
    "businessType": "Manufacturing",
    "contactInfo": "contact@techcorp.com",
    "allocationDate": "2025-09-02"
  }'
```

4. Update Plot Status (Partial Update)
Endpoint: `PATCH /api/v1/plot/update-plot`

Headers:
```bash
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

Request Body:
```json
{
  "country": "benin",
  "zoneCode": "GDIZ",
  "phase": "phase1",
  "plotName": "P2N-65",
  "plotStatus": "Available"
}
```

cURL Command:
```bash
curl -X PATCH "http://20.198.251.4:8000/api/v1/plot/update-plot" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "country": "benin",
    "zoneCode": "GDIZ",
    "phase": "phase1",
    "plotName": "P2N-65",
    "plotStatus": "Available"
```

```
  }'
```

Zone Management Endpoints

1. Create Zone
Endpoint: `POST /api/v1/country/zone`

Headers:
```bash
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

Request Body:
```json
{
  "zoneCode": "GSEZ",
  "zoneName": "Greater Sokoto Economic Zone",
  "country": "gabon",
  "zoneType": "SEZ",
  "landArea": 500.5,
  "description": "Special Economic Zone for manufacturing and trade"
}
```

cURL Command:
```bash
curl -X POST "http://20.198.251.4:8000/api/v1/country/zone" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "zoneCode": "GSEZ",
    "zoneName": "Greater Sokoto Economic Zone",
    "country": "gabon",
    "zoneType": "SEZ",
    "landArea": 500.5,
    "description": "Special Economic Zone for manufacturing and trade"
  }'
```

User Management Endpoints

Note: All user management endpoints require `super_admin` role.

1. Create User
Endpoint: `POST /api/v1/user/create_user`

Headers:
```bash
Authorization: Bearer <SUPER_ADMIN_JWT_TOKEN>
Content-Type: application/json
```

Request Body:
```json
{
  "email": "admin_cameroon@gmail.com",
  "role": "zone_admin",
  "zone": "cameroon"
}
```

cURL Command:
```bash
curl -X POST "http://20.198.251.4:8000/api/v1/user/create_user" \
  -H "Authorization: Bearer <SUPER_ADMIN_JWT_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "email": "admin_cameroon@gmail.com",
    "role": "zone_admin",
    "zone": "cameroon"
  }'
```

2. Update User
Endpoint: `PUT /api/v1/user/update_user`

Headers:
```bash
Authorization: Bearer <SUPER_ADMIN_JWT_TOKEN>
Content-Type: application/json
```

Request Body:
```json
{
  "email": "admin_cameroon@gmail.com",
  "zone": "chad"
```

}
```

**cURL Command**:
```bash
curl -X PUT "http://20.198.251.4:8000/api/v1/user/update_user" \
  -H "Authorization: Bearer <SUPER_ADMIN_JWT_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "email": "admin_cameroon@gmail.com",
    "zone": "chad"
  }'
```

Country-Zone Mappings

Country	Zone Code	Full Name
benin	GDIZ	Gombe Development Industrial Zone
drc	CIP	Cross River Industrial Park
gabon	GSEZ	Greater Sokoto Economic Zone
nigeria	IPR	Industrial Park Rivers
roc	PIC	Plateau Industrial Complex
rwanda	BSEZ	Bayelsa Special Economic Zone
togo	PIA	Plateau Industrial Area
tanzania	SINOTAN	Sino-Tanzania Industrial Zone

Sample Test Sequence

1. Health Check
```bash
curl -X GET "http://20.198.251.4:8000/health"
```

2. Get JWT Token
```bash
curl -X POST "http://20.198.251.4:8000/api/v1/auth/token" \
  -H "Authorization: Secret arise-plot-management-auth-secret-2025" \
  -H "Content-Type: application/json" \
  -d '{
    "userId": "admin_benin@gmail.com",
    "role": "zone_admin",

```
    "zone": "benin"
  }'
```

3. Get Available Plots
```bash
# Replace <JWT_TOKEN> with the token from step 2
curl -X GET "http://20.198.251.4:8000/api/v1/plot/available?
country=benin&zoneCode=GDIZ&phase=phase1" \
  -H "Authorization: Bearer <JWT_TOKEN>"
```

4. Get Plot Details
```bash
curl -X GET "http://20.198.251.4:8000/api/v1/plot/plot-detail?
country=benin&zoneCode=GDIZ&phase=phase1" \
  -H "Authorization: Bearer <JWT_TOKEN>"
```

5. Update Plot
```bash
curl -X PUT "http://20.198.251.4:8000/api/v1/plot/update-plot" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "country": "benin",
    "zoneCode": "GDIZ",
    "phase": "phase1",
    "plotName": "P1A-01",
    "plotStatus": "Occupied",
    "businessName": "Test Company",
    "businessType": "Manufacturing"
  }'
```

Error Handling

Common Error Responses

401 Unauthorized:
```json
{
  "detail": {
    "error_code": "MISSING_AUTHORIZATION",
```

```
    "message": "Authorization header is required",
    "details": {"expected_format": "Authorization: Secret <secret-key>"}
  }
}
```

403 Forbidden:
```json
{
  "detail": {
    "error_code": "ACCESS_DENIED",
    "message": "You can only update plots in zone GDIZ (country: benin)",
    "details": {
      "user_zone": "benin",
      "requested_zone": "PIA"
    }
  }
}
```

404 Not Found:
```json
{
  "detail": {
    "error_code": "PLOT_NOT_FOUND",
    "message": "Plot P1A-01 not found in benin/GDIZ/phase1"
  }
}
```

Notes

1. **JWT Token Expiry**: Tokens expire after 24 hours.
2. **Zone Restrictions**: Zone admins can only modify plots in their assigned zone.
3. **Pagination**: Use the `cursor` parameter for pagination (check response headers).
4. **Rate Limiting**: Currently not implemented.
5. **CORS**: Configured for localhost development.
6. **Firebase**: Uses Firestore as the database backend.

Quick Test Script

Save this as `test_api.sh`:

```bash
#!/bin/bash

BASE_URL="http://20.198.251.4:8000"
API_BASE="$BASE_URL/api/v1"

echo "=== Health Check ==="
curl -s "$BASE_URL/health" | jq

echo -e "\n=== Get JWT Token ==="
TOKEN_RESPONSE=$(curl -s -X POST "$API_BASE/auth/token" \
  -H "Authorization: Secret arise-plot-management-auth-secret-2025" \
  -H "Content-Type: application/json" \
  -d '{
    "userId": "admin_benin@gmail.com",
    "role": "zone_admin",
    "zone": "benin"
  }')

echo "$TOKEN_RESPONSE" | jq

# Extract token
TOKEN=$(echo "$TOKEN_RESPONSE" | jq -r '.access_token')

echo -e "\n=== Get Available Plots ==="
curl -s -X GET "$API_BASE/plot/available?
country=benin&zoneCode=GDIZ&phase=phase1&limit=5" \
  -H "Authorization: Bearer $TOKEN" | jq

echo -e "\n=== Get Plot Details ==="
curl -s -X GET "$API_BASE/plot/plot-detail?
country=benin&zoneCode=GDIZ&phase=phase1&limit=5" \
  -H "Authorization: Bearer $TOKEN" | jq
```

Make it executable and run:
```bash
chmod +x test_api.sh
./test_api.sh
```
</file>

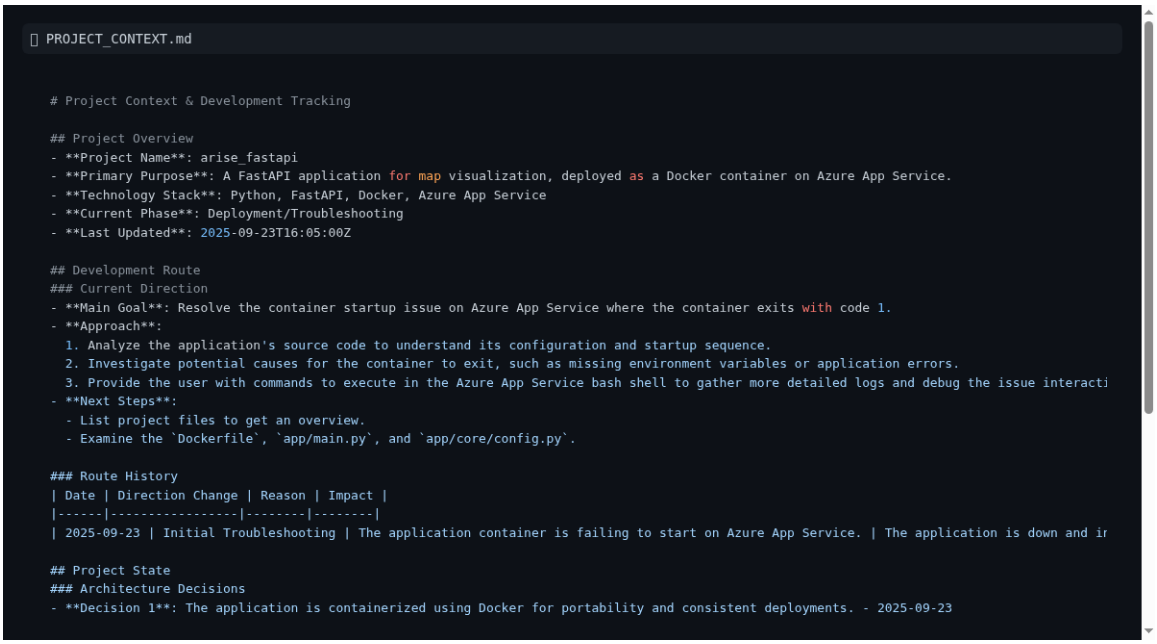
<file path="Arise FastAPI Backend.postman_collection_Azure_UAT">

```
{
        "info": {
```

```
		"_postman_id": "45cf3c4b-8a65-4cfe-bfc5-07858dadda00",
		"name": "Arise FastAPI Backend",
		"schema":
"https://schema.getpostman.com/json/collection/v2.1.0/collection.json",
		"_exporter_id": "42887775"
	},
	"item": [
		{
			"name": "auth/token",
			"request": {
				"method": "POST",
				"header": [
					{
						"key": "Cache-Control",
						"value": "no-cache",
						"type": "text",
						"disabled": true
					},
					{
						"key": "Postman-Token",
						"value": "<calculated when request is
sent>",
						"type": "text",
						"disabled": true
					},
					{
						"key": "Content-Type",
						"value": "application/json",
						"type": "text",
						"disabled": true
					},
					{
						"key": "Content-Length",
						"value": "<calculated when request is
sent>",
						"type": "text",
						"disabled": true
					},
					{
						"key": "Host",
						"value": "<calculated when request is
sent>",
						"type": "text",
						"disabled": true
```

```
},
{
        "key": "User-Agent",
        "value": "PostmanRuntime/7.39.1",
        "type": "text",
        "disabled": true
},
{
        "key": "Accept",
        "value": "*/*",
        "type": "text",
        "disabled": true
},
{
        "key": "Accept-Encoding",
        "value": "gzip, deflate, br",
        "type": "text",
        "disabled": true
},
{
        "key": "Connection",
        "value": "keep-alive",
        "type": "text",
        "disabled": true
},
{
        "key": "Authorization",
        "value": "Secret arise-
```


Figure: Usage Guide overview



```
 PROJECT_CONTEXT.md


# Project Context & Development Tracking

## Project Overview
- **Project Name**: arise_fastapi
- **Primary Purpose**: A FastAPI application for map visualization, deployed as a Docker container on Azure App Service.
- **Technology Stack**: Python, FastAPI, Docker, Azure App Service
- **Current Phase**: Deployment/Troubleshooting
- **Last Updated**: 2025-09-23T16:05:00Z

## Development Route
### Current Direction
- **Main Goal**: Resolve the container startup issue on Azure App Service where the container exits with code 1.
- **Approach**:
  1. Analyze the application's source code to understand its configuration and startup sequence.
  2. Investigate potential causes for the container to exit, such as missing environment variables or application errors.
  3. Provide the user with commands to execute in the Azure App Service bash shell to gather more detailed logs and debug the issue interacti
- **Next Steps**:
  - List project files to get an overview.
  - Examine the `Dockerfile`, `app/main.py`, and `app/core/config.py`.

### Route History
| Date | Direction Change | Reason | Impact |
|------|----------------|--------|--------|
| 2025-09-23 | Initial Troubleshooting | The application container is failing to start on Azure App Service. | The application is down and ir

## Project State
### Architecture Decisions
- **Decision 1**: The application is containerized using Docker for portability and consistent deployments. - 2025-09-23
```

Authentication and Authorization

Authentication and Authorization

This section outlines the security mechanisms governing access to the Arise Plot Management API. It details how users are authenticated to verify their identity and how authorization rules are applied to determine their permitted actions.

Authentication

The API utilizes JSON Web Tokens (JWT) for authenticating users. Upon successful verification of user credentials and associated role/zone information, a JWT is issued. This token serves as proof of identity for subsequent API requests.

Process Overview:

1. **Token Generation:** A `POST` request to the `/api/v1/auth/token` endpoint is used to obtain a JWT. This request requires an `Authorization` header containing a secret key and a JSON payload specifying the `userId`, `role`, and `zone`.
2. **Token Usage:** The obtained `access_token` must be included in the `Authorization: Bearer <JWT_TOKEN>` header for all protected API endpoints.

This approach ensures that only authenticated users can access the system, and their identity is securely validated with each request.

Authorization

Authorization is managed through role-based access control (RBAC) and zone-specific permissions. The API enforces that users can only perform actions within the scope of their assigned role and the geographical zones they are authorized to manage.

Key Principles:

Role-Based Permissions: Different roles (e.g., `zone_admin`, `super_admin`) are granted specific sets of permissions. For instance, `super_admin` roles are required for user management operations.
Zone Restrictions: Users with `zone_admin` roles are restricted to performing operations only within their designated `zone`. Any attempt to access or modify data outside their authorized zone will result in a `403 Forbidden` error.
API Enforcement: Authorization checks are performed at the API endpoint level to ensure that access control policies are strictly adhered to.

This layered security model ensures data integrity and prevents unauthorized access or modifications, maintaining a secure and controlled environment for plot management operations.

Plot Management Operations

Plot Management Operations

This section details the operational capabilities of the Arise Plot Management API, focusing on how users interact with plot data through defined endpoints. It covers the core functionalities for retrieving, updating, and managing plot information within the system.

Plot Data Retrieval

The API provides endpoints to query available plots and retrieve detailed information about specific plots. These operations are crucial for visualizing and understanding the current state of land plots within designated zones.

Available Plots: Users can fetch a list of plots that are currently available for allocation or development. This function allows for filtering by country, zone, and phase, with support for pagination to manage large datasets efficiently.
Plot Details: Detailed information for individual plots can be accessed, offering insights into plot status, associated businesses, contact information, and allocation dates. This endpoint also supports pagination and filtering parameters.

These retrieval operations are designed to provide stakeholders with real-time access to plot inventory, supporting strategic planning and operational decision-making.

Plot Data Updates

The API supports both complete and partial updates to plot information, allowing for dynamic management of plot statuses and associated details.

Complete Plot Update: This operation allows for a full modification of a plot's record. It requires all relevant fields, including country, zone code, phase, plot name, status, business name, business type, contact information, and allocation date, to be provided. This is useful when significant changes occur, such as a new business occupying a plot or a change in its development status.

Partial Plot Update: For more granular changes, a `PATCH` method is available to update specific fields, such as the `plotStatus`. This is ideal for quick adjustments without needing to resubmit the entire plot record, ensuring agility in operations.

These update functionalities are critical for maintaining an accurate and up-to-date representation of plot utilization and status within the Arise platform.

Zone Management Operations

Zone Management Operations

This section details the capabilities for managing geographical zones within the Arise Plot Management system. It outlines the processes for creating and managing these zones, which serve as foundational structures for plot allocation and administration.

Zone Creation and Configuration

The API provides functionality to define and register new economic zones within the system. This process involves specifying key attributes that characterize each zone, ensuring comprehensive data management for different geographical and developmental areas.

Endpoint: `POST /api/v1/country/zone`

Purpose: To establish new economic zones within the Arise platform.

Key Attributes: When creating a zone, essential details such as the `zoneCode`, `zoneName`, `country`, `zoneType` (e.g., SEZ), `landArea`, and a descriptive `description` are provided. This allows for precise categorization and management of diverse economic zones.

The ability to create and configure zones programmatically ensures that the platform can adapt to evolving geographical and administrative structures, providing a robust framework for plot management across various regions. The system supports a defined set of country-zone mappings, as detailed in the API documentation.

Zone Administration and Access Control

Zone management is intrinsically linked to user access control. The system employs a role-based access control (RBAC) model where users, particularly `zone_admin` roles, are assigned to specific zones. This ensures that administrative actions related to plots and other zone-specific data are confined to the authorized geographical boundaries.

Role-Based Permissions: Users with the `zone_admin` role are granted permissions to manage plots and resources exclusively within their assigned zone.
Super Admin Privileges: A `super_admin` role possesses broader permissions, including the ability to manage users and their zone assignments, facilitating centralized oversight and control over the entire zone hierarchy.

This operational structure ensures that zone administration is secure, efficient, and aligned with the geographical responsibilities of each user role.

User Management Operations

User Management Operations

This section details the capabilities for managing user accounts and their associated permissions within the Arise Plot Management system. It outlines the essential operations for creating, updating, and controlling user access, ensuring secure and efficient administration of the platform.

User Account Creation

The API provides a dedicated endpoint for the creation of new user accounts. This process is restricted to users with `super_admin` privileges, ensuring centralized control over user onboarding. When creating a user, essential details such as their email address, assigned role, and the specific zone they will manage are specified. This granular approach to user provisioning allows for precise definition of access rights from the outset.

User Profile Updates

Existing user accounts can be modified to reflect changes in roles or zone assignments. The `PUT /api/v1/user/update_user` endpoint enables

administrators to update user profiles, ensuring that user permissions remain current and aligned with organizational requirements. This capability is crucial for maintaining an accurate representation of user access levels and responsibilities within the platform.

Role-Based Access Control (RBAC)

User management is intrinsically tied to the system's role-based access control (RBAC) model. Users are assigned specific roles (e.g., `zone_admin`, `super_admin`), which dictate their permissions and the scope of their operations. `zone_admin` users are restricted to managing plots and resources within their assigned zone, while `super_admin` users possess broader privileges, including the management of other users and their assignments. This ensures a secure and hierarchical access structure, preventing unauthorized access to sensitive data and functionalities.

Architecture and Design

Architecture and Design

This section provides a high-level overview of the Arise Plot Management API's architecture and design principles, focusing on its core components, data flow, and key technological choices. It aims to inform business stakeholders and technical managers about the system's structure and its suitability for meeting project objectives.

System Overview

The Arise Plot Management API is a backend service built using Python and the FastAPI framework. Its primary function is to manage plot data, facilitate zone administration, and handle user authentication and authorization. The application is designed to be containerized using Docker, enabling consistent deployment across various environments, including cloud platforms like Azure App Service. This architecture prioritizes scalability, security, and efficient data handling for geographical plot management.

Core Components

The API's architecture is composed of several interconnected modules, each responsible for specific functionalities:

API Layer (FastAPI): This layer handles incoming HTTP requests, routes them to appropriate services, and formats responses. It defines the API endpoints for all operations, including authentication, plot management, and zone administration. The use of FastAPI ensures high performance and

automatic generation of interactive API documentation.

Core Services: This module contains the business logic for the application. It orchestrates operations such as user authentication, authorization checks, and data processing for plot and zone management. Key services include authentication handling, which manages JWT generation and validation, and data access logic for interacting with the database.

Data Storage (Firebase Firestore): The application leverages Firebase Firestore as its primary database. Firestore is a NoSQL cloud database that offers real-time data synchronization, scalability, and flexibility for storing structured and unstructured data. This choice supports the need for dynamic data management and accessibility across different geographical zones and plot attributes.

Caching (Redis): For performance optimization, Redis is employed as an in-memory data store for caching frequently accessed data. This reduces database load and speeds up response times for common queries, enhancing the overall user experience.

Data Flow and Security

Requests to the API are initiated through API endpoints. User authentication is performed using JWTs obtained via the `/api/v1/auth/token` endpoint. Once authenticated, authorization checks are enforced based on user roles and assigned zones, ensuring that users only access resources they are permitted to manage. All sensitive data, including authentication secrets, is managed securely, with plans to integrate with Azure Key Vault for production environments. Data operations are performed against Firebase Firestore, with Redis caching for performance.

Deployment Strategy

The Arise Plot Management API is designed for deployment as a Docker container. This containerized approach ensures consistency and portability, allowing the application to be deployed on various infrastructure platforms. Key deployment targets include:

Azure App Service: A managed platform-as-a-service (PaaS) offering that simplifies the deployment and scaling of web applications and containerized services.

Azure Container Apps: A serverless container service that provides a cost-effective and scalable environment for microservices and containerized applications.

Azure Kubernetes Service (AKS): A managed Kubernetes service for orchestrating containerized applications at enterprise scale, suitable for complex microservice architectures.

This multi-option deployment strategy provides flexibility to choose the most appropriate hosting environment based on performance, scalability, and cost requirements.

Key Technologies

Python: The primary programming language, offering a rich ecosystem of libraries for web development and data management.

FastAPI: A modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. Its documentation highlights its speed and ease of use.

Docker: A platform for developing, shipping, and running applications in containers, ensuring consistent execution across development, testing, and production environments.

Firebase Firestore: A NoSQL cloud database from Google, providing real-time data synchronization and scalability. More information can be found at https://firebase.google.com/docs/firestore.

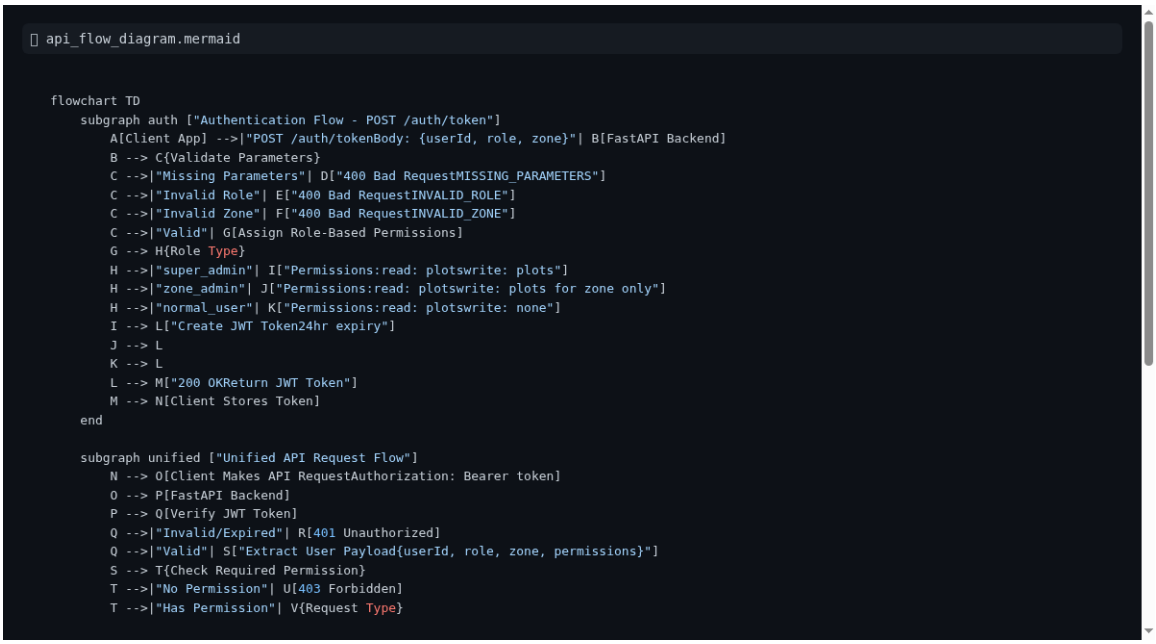
Redis: An open-source, in-memory data structure store, used as a database, cache, and message broker. Its homepage is https://redis.io/.

Azure App Service: A cloud computing service for hosting web applications, REST APIs, and mobile back ends. Details are available at https://azure.microsoft.com/en-us/services/app-service/.

Documentation and Guidelines

Comprehensive API documentation is generated automatically by FastAPI and is accessible via the `/docs` endpoint. This includes interactive guides for testing API requests. Additional guidelines for API usage and development practices are documented in files like `API_REQUESTS_GUIDE.md` and `AGENT-COMMAND-RULES.md`.

Figure: Architecture and Design overview



```
api_flow_diagram.mermaid


flowchart TD
    subgraph auth ["Authentication Flow - POST /auth/token"]
        A[Client App] -->|"POST /auth/tokenBody: {userId, role, zone}"| B[FastAPI Backend]
        B --> C{Validate Parameters}
        C -->|"Missing Parameters"| D["400 Bad RequestMISSING_PARAMETERS"]
        C -->|"Invalid Role"| E["400 Bad RequestINVALID_ROLE"]
        C -->|"Invalid Zone"| F["400 Bad RequestINVALID_ZONE"]
        C -->|"Valid"| G[Assign Role-Based Permissions]
        G --> H{Role Type}
        H -->|"super_admin"| I["Permissions:read: plotswrite: plots"]
        H -->|"zone_admin"| J["Permissions:read: plotswrite: plots for zone only"]
        H -->|"normal_user"| K["Permissions:read: plotswrite: none"]
        I --> L["Create JWT Token24hr expiry"]
        J --> L
        K --> L
        L --> M["200 OKReturn JWT Token"]
        M --> N[Client Stores Token]
    end

    subgraph unified ["Unified API Request Flow"]
        N --> O[Client Makes API RequestAuthorization: Bearer token]
        O --> P[FastAPI Backend]
        P --> Q[Verify JWT Token]
        Q -->|"Invalid/Expired"| R[401 Unauthorized]
        Q -->|"Valid"| S["Extract User Payload{userId, role, zone, permissions}"]
        S --> T{Check Required Permission}
        T -->|"No Permission"| U[403 Forbidden]
        T -->|"Has Permission"| V{Request Type}
```

API Design Principles

API Design Principles

This section outlines the core principles guiding the design and development of the Arise Plot Management API. Adhering to these principles ensures consistency, maintainability, scalability, and a positive developer experience for both internal teams and external consumers of the API.

Consistency and Predictability

Uniform Naming Conventions: All API endpoints, parameters, and response fields follow a consistent naming scheme (e.g., snake_case for Python, camelCase for JSON). This predictability reduces the learning curve for developers interacting with the API.

Standardized HTTP Methods: RESTful principles are applied, using appropriate HTTP methods (GET, POST, PUT, PATCH, DELETE) for their intended operations.

Consistent Error Handling: A standardized error response format is used across all endpoints, providing clear and actionable information about issues encountered during request processing.

Scalability and Performance

Stateless Design: API endpoints are designed to be stateless, meaning each request contains all the necessary information for processing. This facilitates horizontal scaling by allowing any instance of the API to handle

any request.

Pagination: For endpoints that return lists of resources, pagination is implemented using cursors or offset/limit parameters to manage large datasets efficiently and prevent performance degradation.

Caching Strategies: Where appropriate, caching mechanisms (e.g., Redis) are employed to store frequently accessed data, reducing database load and improving response times.

Security and Robustness

Authentication and Authorization: A robust authentication mechanism (JWT-based) is in place to verify user identities, and role-based access control (RBAC) ensures that users can only access resources and perform actions permitted by their assigned roles and zones.

Input Validation: All incoming data is rigorously validated against predefined schemas to prevent invalid or malicious data from entering the system.

Secure Secret Management: Sensitive credentials and keys are managed securely, with a plan to integrate with Azure Key Vault for production environments to avoid hardcoding secrets.

Maintainability and Developer Experience

Self-Documenting API: Leveraging FastAPI, the API generates interactive documentation (Swagger UI/OpenAPI) accessible at `/docs`. This documentation serves as a single source of truth for API consumers.

Clear Versioning: API versioning (e.g., `/api/v1`) is implemented to manage changes and ensure backward compatibility for existing integrations.

Modular Design: The codebase is structured into logical modules (e.g., `app/api`, `app/core`, `app/services`), promoting code reusability and simplifying maintenance.

By adhering to these principles, the Arise Plot Management API aims to provide a reliable, secure, and user-friendly interface for managing plot and zone data.

Data Models and Schemas

Data Models and Schemas

This section details the data models and schemas that define the structure and integrity of the data managed by the Arise Plot Management API. These models are crucial for ensuring data consistency, enabling efficient data retrieval, and facilitating seamless integration with various system components and external applications.

Data Model Overview

The Arise Plot Management API utilizes a structured approach to data representation, primarily leveraging **Pydantic models** within its `app/schemas` directory. These models serve as the blueprint for data exchanged between the API, its services, and the underlying data store (Firebase Firestore). They define the expected attributes, data types, and validation rules for key entities such as users, plots, and zones.

Key Data Entities

The core data entities and their corresponding schemas are designed to capture essential information for plot and zone management:

- **User Schemas (`app/schemas/auth.py`):** Define the structure for user authentication requests and responses. This includes fields for user identification, roles, and associated zones, forming the basis for role-based access control (RBAC).
- **Plot Schemas (`app/schemas/plots.py`):** Detail the attributes of a plot, such as its name, status, location (country, zone, phase), business association, and contact information. These schemas ensure that plot data is consistently recorded and managed.
- **Zone Schemas (`app/schemas/zones.py`):** Define the structure for zone-related data, including zone codes, names, country, land area, and descriptive information. This supports the hierarchical organization of plots within specific geographical or administrative zones.

Purpose and Benefits

- **Data Integrity:** Pydantic models enforce data types and validation rules, preventing malformed data from entering the system and ensuring data accuracy.
- **API Contract:** These schemas act as a clear contract for API interactions, defining the expected input and output formats for requests and responses. This simplifies integration for developers and downstream systems.
- **Code Readability and Maintainability:** By centralizing data definitions, schemas improve code clarity and make it easier to understand and maintain the data structures used throughout the application.
- **Foundation for Services:** The defined schemas provide a robust foundation for the API's services (`app/services`), enabling them to process and manipulate data reliably.

By employing well-defined data models and schemas, the Arise Plot Management API ensures a structured, reliable, and maintainable approach to data management, which is fundamental for its operational efficiency and scalability.

Core Components

Core Components

This section outlines the fundamental building blocks of the Arise Plot Management API, describing their roles, benefits, and how they contribute to the overall system functionality.

Application Architecture

The Arise Plot Management API is built using a modern, scalable, and maintainable architecture. At its heart is a **FastAPI** web framework, chosen for its high performance, ease of use, and automatic interactive API documentation generation. This framework efficiently handles incoming requests, routes them to the appropriate services, and formats the responses.

The application is designed with clear separation of concerns:

API Layer (`app/api`): This layer exposes the public-facing endpoints of the API. It handles request validation, authentication, and authorization, then delegates the core business logic to the service layer. Key functionalities include user authentication, plot management, and zone administration.
Service Layer (`app/services`): This layer contains the core business logic. It orchestrates operations such as user authentication, authorization checks, and data processing for plot and zone management. It interacts with the data storage and caching layers.
Data Storage (Firebase Firestore): The application leverages Firebase Firestore as its primary NoSQL cloud database. This choice supports real-time data synchronization, scalability, and flexibility for managing plot and zone data across different regions. More details can be found at https://firebase.google.com/docs/firestore.
Caching (Redis): To enhance performance and reduce database load, Redis is employed as an in-memory data store for caching frequently accessed information. This significantly speeds up response times for common queries. The official Redis website is https://redis.io/.

This modular design ensures that each component can be developed, tested, and scaled independently, contributing to the overall robustness and maintainability of the system.

Key Technologies

The Arise Plot Management API is developed using a robust and modern technology stack:

Python: The primary programming language, providing a rich ecosystem for web development and data management.
FastAPI: A high-performance, modern web framework for building APIs with Python 3.7+, known for its speed and automatic interactive documentation. Learn more at https://fastapi.tiangolo.com/.
Docker: This containerization platform ensures that the application runs consistently across development, testing, and production environments. It is essential for deployment on cloud platforms.
Firebase Firestore: A scalable NoSQL cloud database used for data persistence. Details are available at https://firebase.google.com/docs/firestore.
Redis: An in-memory data structure store used for caching, significantly improving application performance. More information can be found at https://redis.io/.
Azure Services: The application is designed for deployment on Azure, with specific services like Azure App Service, Azure Container Apps, or Azure Kubernetes Service (AKS) being primary targets. Azure App Service offers a managed platform for web applications and APIs: https://azure.microsoft.com/en-us/services/app-service/.

This combination of technologies provides a powerful, flexible, and scalable foundation for the Arise Plot Management API.

Deployment and Security

The Arise Plot Management API is containerized using Docker, enabling consistent deployment across various cloud infrastructures. The primary target for production deployment is **Azure App Service**, a fully managed platform-as-a-service (PaaS) that simplifies the deployment and scaling of web applications and containerized services. Other viable options include Azure Container Apps for serverless deployments or Azure Kubernetes Service (AKS) for complex orchestration needs.

Security is paramount. The API employs **JWT-based authentication** for verifying user identities and **role-based access control (RBAC)** to enforce permissions based on user roles and assigned zones. Input validation is performed rigorously to protect against malicious data. Sensitive information, such as authentication secrets, will be managed securely, with

plans to integrate with **Azure Key Vault** for production environments to avoid hardcoding credentials.

The API versioning (e.g., `/api/v1`) ensures backward compatibility and allows for controlled evolution of the API. Comprehensive, interactive documentation is automatically generated by FastAPI and accessible at the `/docs` endpoint, serving as a crucial resource for developers and stakeholders.

Caching Strategies

Caching Strategies

To optimize performance and reduce the load on our primary data store (Firebase Firestore), the Arise Plot Management API implements strategic caching. This involves storing frequently accessed data in a fast, in-memory data store, Redis.

Core Caching Approach

The API leverages Redis to cache essential data that is read frequently but changes infrequently. This includes:

- **User Permissions and Roles:** Information about user roles and their associated permissions is cached to speed up authorization checks on incoming API requests.
- **Zone and Country Configurations:** Static or semi-static configuration data related to countries and zones is cached to avoid repeated database lookups.
- **Frequently Accessed Plot Data:** For common queries, such as listing available plots within a specific region, results can be cached to significantly reduce latency.

Benefits of Caching

- **Improved Response Times:** By serving data from the in-memory Redis cache, API response times are dramatically reduced, leading to a more responsive user experience.
- **Reduced Database Load:** Caching offloads a significant portion of read operations from Firebase Firestore, conserving database resources and potentially lowering costs.
- **Enhanced Scalability:** A well-implemented caching strategy allows the API to handle a higher volume of concurrent requests more efficiently.
- **Increased Availability:** In scenarios where database connectivity might be temporarily impaired, cached data can still be served, providing a degree of resilience.

Implementation Details

The caching logic is managed within the `app/services/cache_strategies.py` module. This ensures that caching is applied consistently across relevant API operations. Cache invalidation strategies are employed to ensure that stale data is not served, typically by invalidating cache entries when the underlying data is modified.

For more information on Redis, please refer to the official documentation: https://redis.io/docs/.