Arise Mapview: FastAPI App 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

# 1. Introduction

# 1. Introduction

This document provides a consolidated view of the Arise Plot Management API codebase, designed for comprehensive analysis by AI systems. It serves as a single source of truth for understanding the project's structure, content, and purpose.

## Purpose

The primary purpose of this packed codebase representation is to facilitate efficient and accurate analysis, code reviews, and other automated processes by AI models. By merging the entire repository into a single, structured document, it ensures that AI systems have complete context for tasks such as understanding functionality, identifying potential issues, and generating insights.

## File Format and Content

This file is organized into distinct sections for clarity and ease of processing:

1. **File Summary**: An overview of this document's purpose, format, usage, and notes.
2. **Repository Information**: High-level details about the repository.
3. **Directory Structure**: A hierarchical view of the project's folders and files.
4. **Repository Files**: The full content of each file within the repository (excluding binary files and items excluded by `.gitignore`).

Each file entry includes its full path and its complete content, enabling AI systems to reference and analyze individual components within the context of the entire project.

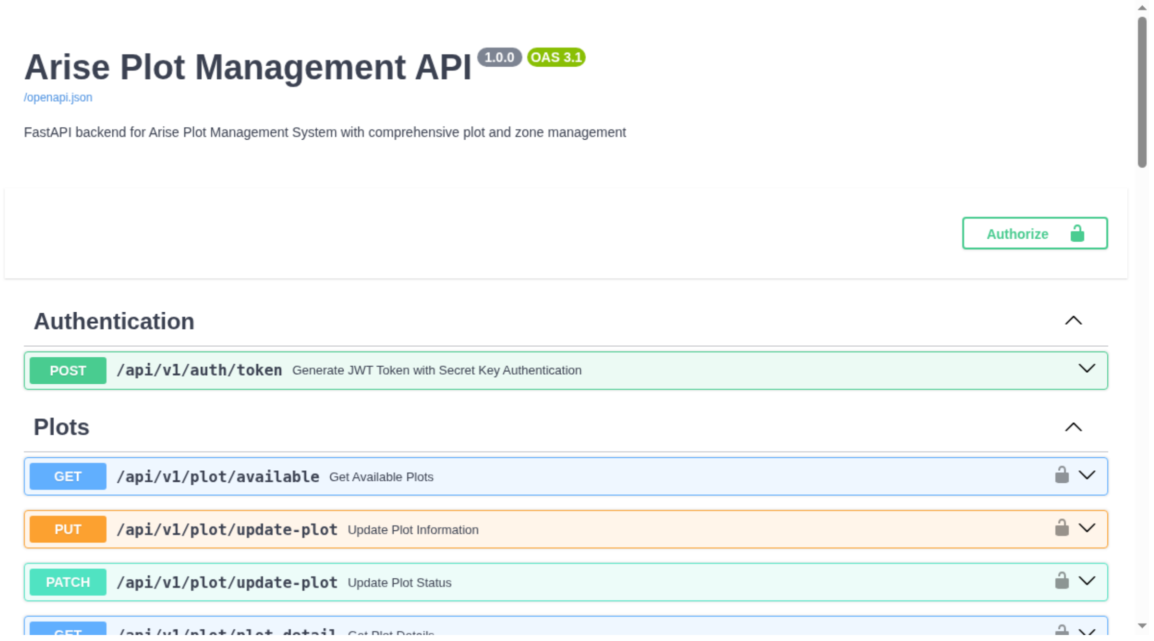
## Usage Guidelines

This document is intended for read-only access. All modifications should be made to the original source files within the repository. When processing this file, the file path attribute is crucial for distinguishing between different code components. As this file may contain sensitive information, it must be handled with the same security protocols as the original codebase.

## Notes

Certain files have been excluded from this representation based on `.gitignore` rules and Repomix's configuration. Binary files are not included; their paths are available in the Repository Structure section. Files matching ignore patterns are omitted to maintain focus on source code and configuration. Files are sorted by Git change count, with more frequently modified files appearing towards the end.

Figure: Application Screenshot: swagger_docs



Arise Plot Management API 1.0.0 OAS 3.1

/openapi.json

FastAPI backend for Arise Plot Management System with comprehensive plot and zone management

Authentication

POST /api/v1/auth/token Generate JWT Token with Secret Key Authentication

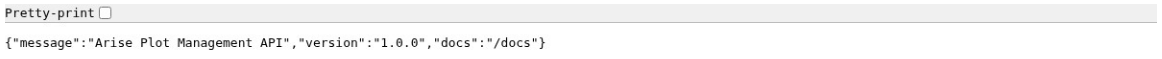
Plots

GET /api/v1/plot/available Get Available Plots

PUT /api/v1/plot/update-plot Update Plot Information

PATCH /api/v1/plot/update-plot Update Plot Status

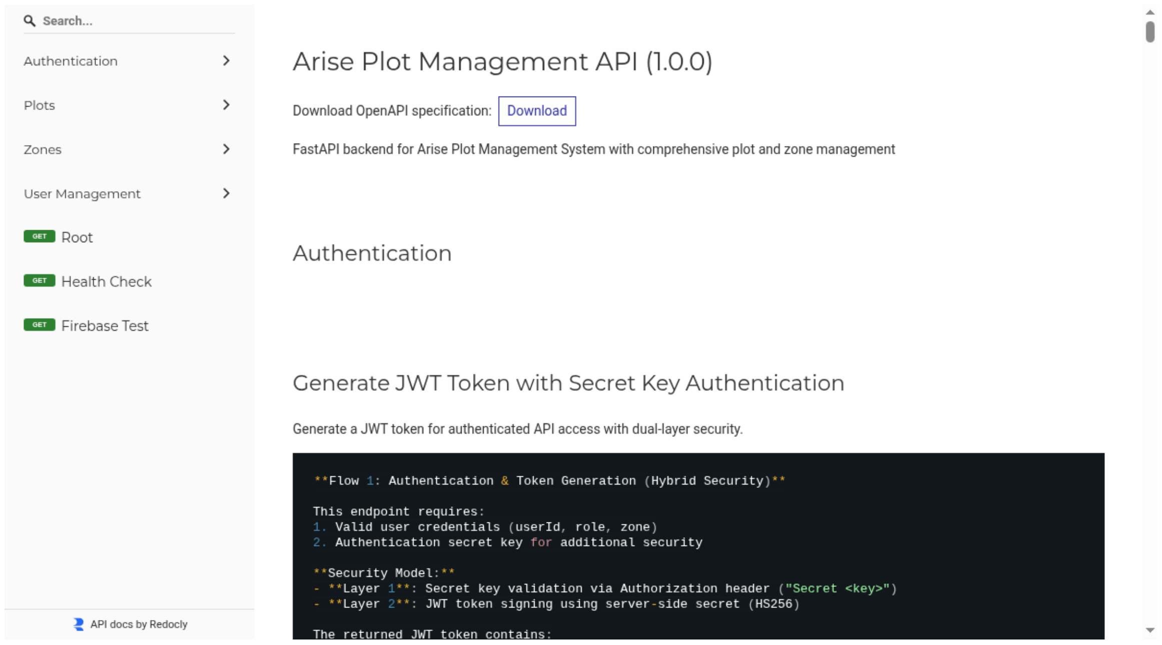
Figure: Application Screenshot: api_home



Pretty-print

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

Figure: Application Screenshot: redoc



Arise Plot Management API (1.0.0)

Download OpenAPI specification: Download

FastAPI backend for Arise Plot Management System with comprehensive plot and zone management

Authentication

Generate JWT Token with Secret Key Authentication

Generate a JWT token for authenticated API access with dual-layer security.

```
**Flow 1: Authentication & Token Generation (Hybrid Security)**

This endpoint requires:
1. Valid user credentials (userId, role, zone)
2. Authentication secret key for additional security

**Security Model:**
- **Layer 1**: Secret key validation via Authorization header ("Secret <key>")
- **Layer 2**: JWT token signing using server-side secret (HS256)

The returned JWT token contains:
```

## 2. Project Overview

# 2. Project Overview

This repository contains the codebase for the Arise Plot Management API, a FastAPI application designed for managing land plot data across multiple countries and zones. The application facilitates operations related to plot availability, details, status updates, and zone management. It is built with Python and FastAPI, leveraging Firebase Firestore for data persistence and designed for deployment as a Docker container.

### Purpose

The Arise Plot Management API provides a robust backend for systems requiring detailed management and visualization of land plots. It enables authorized users to perform critical operations such as:

Authentication: Secure access through JWT token generation based on user roles and zones.
Plot Management: Querying available plots, retrieving plot details, and updating plot information (status, allocation, business details).
Zone Management: Creating and managing geographical zones within countries, defining their attributes and purpose.
User Management: Administering user accounts, assigning roles, and managing zone access.

This API is engineered for scalability and reliability, supporting multi-country operations and role-based access control to ensure data integrity and security.

### Key Capabilities and Benefits

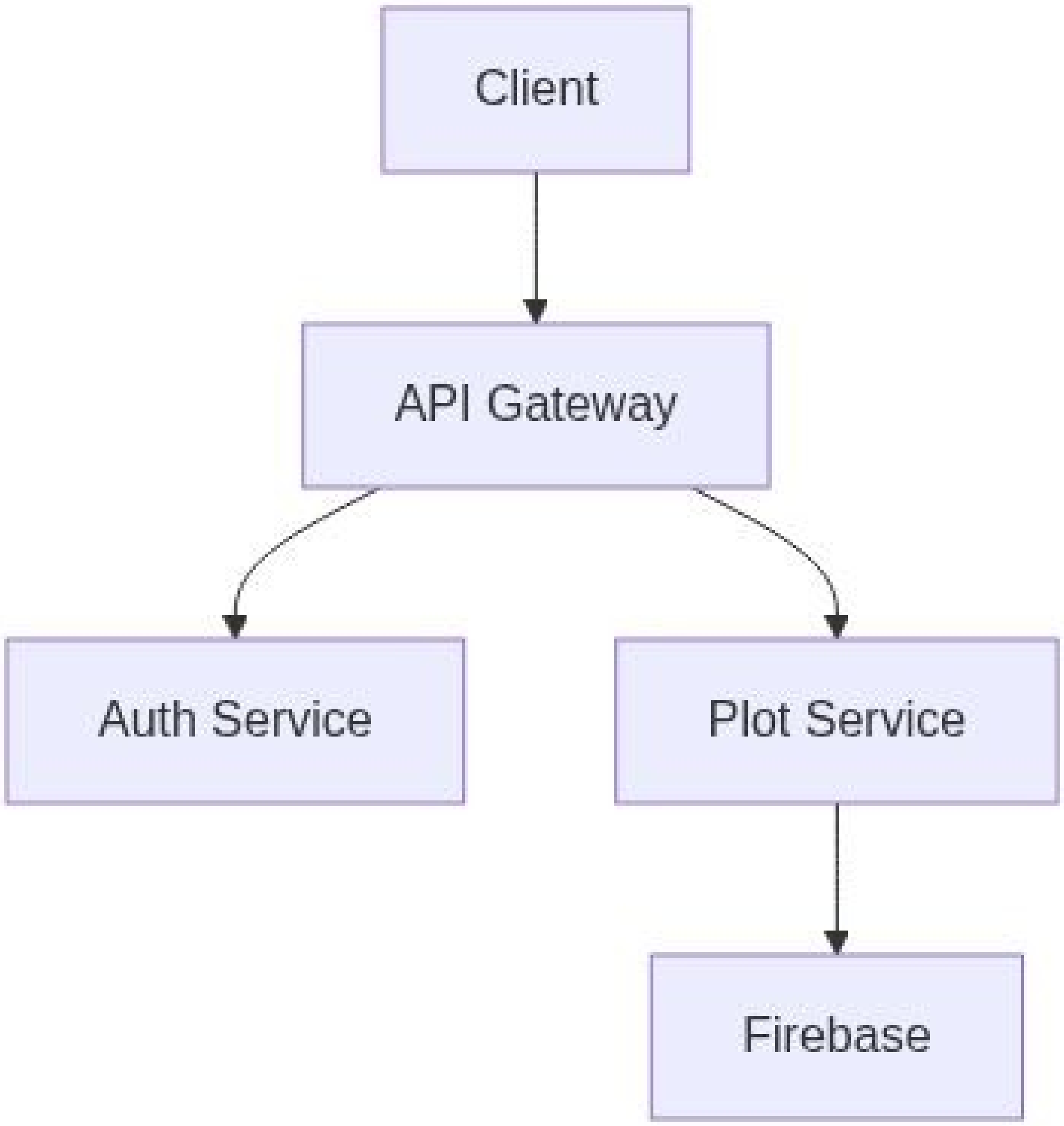
Centralized Plot Data Management: A single source of truth for all plot-related information, accessible via a well-defined API.
Role-Based Access Control (RBAC): Ensures that users can only access and modify data relevant to their assigned roles and geographical zones, enhancing security and data governance.
Scalable Architecture: Built on FastAPI and designed for containerization, allowing for efficient deployment and scaling on cloud platforms like Azure.
Integration Ready: Designed to integrate with other systems through its RESTful API, providing flexible data access and management capabilities.
Comprehensive Documentation: Includes detailed API request guides and client meeting solutions documents to facilitate understanding and adoption.

### Technology Stack

Language: Python
Framework: FastAPI (for building the API)
Database: Firebase Firestore (for data storage)
Containerization: Docker
Deployment Target: Azure App Service (or similar container orchestration platforms)

This project aims to provide a secure, efficient, and scalable solution for managing complex land plot data, supporting strategic development and operational planning.

Figure: Architecture Diagram: 2. Project Overview



## 3. Architecture & Design

## 3. Architecture & Design

The Arise Plot Management API is architected as a robust, containerized backend service designed for efficient management of land plot data across various geographical regions. Its core functionality is exposed via a RESTful API built with Python and the FastAPI framework, ensuring high performance and developer productivity.

### Core Components

The system is structured into modular components to promote maintainability and scalability:

API Layer: Handles incoming HTTP requests, request validation, and routing to appropriate services. This layer is responsible for exposing the core functionalities such as plot and zone management, user authentication, and user management.
Service Layer: Contains the business logic for each functional area. Services interact with the data layer to perform operations like retrieving plot details, updating plot statuses, or creating new zones. Key services include `PlotService`, `ZoneService`, and `UserService`.
Data Layer: Manages data persistence. The primary data store is Firebase Firestore, a NoSQL cloud database, which provides a scalable and flexible solution for storing diverse plot and zone information. Integration with Firebase is handled through dedicated utility modules.
Authentication Module: Implements secure user authentication and authorization. It generates and validates JSON Web Tokens (JWT) based on user roles and assigned zones, enforcing access control across the API.

### Design Principles

The architecture adheres to several key design principles:

Modularity: Components are loosely coupled, allowing for independent development, testing, and deployment.
Scalability: Designed to handle increasing loads through horizontal scaling of containerized instances and leveraging the scalability of Firebase Firestore.
Security: Implements role-based access control (RBAC) and secure authentication mechanisms to protect sensitive data.
Maintainability: Clear separation of concerns and well-defined interfaces facilitate easier updates and debugging.
Containerization: The application is packaged as a Docker container, ensuring consistent deployment across different environments, from development to production on platforms like Azure App Service.

This architectural approach ensures that the Arise Plot Management API is a reliable, secure, and scalable solution capable of supporting complex land management operations.

Figure: Architecture Diagram: 3. Architecture & Design

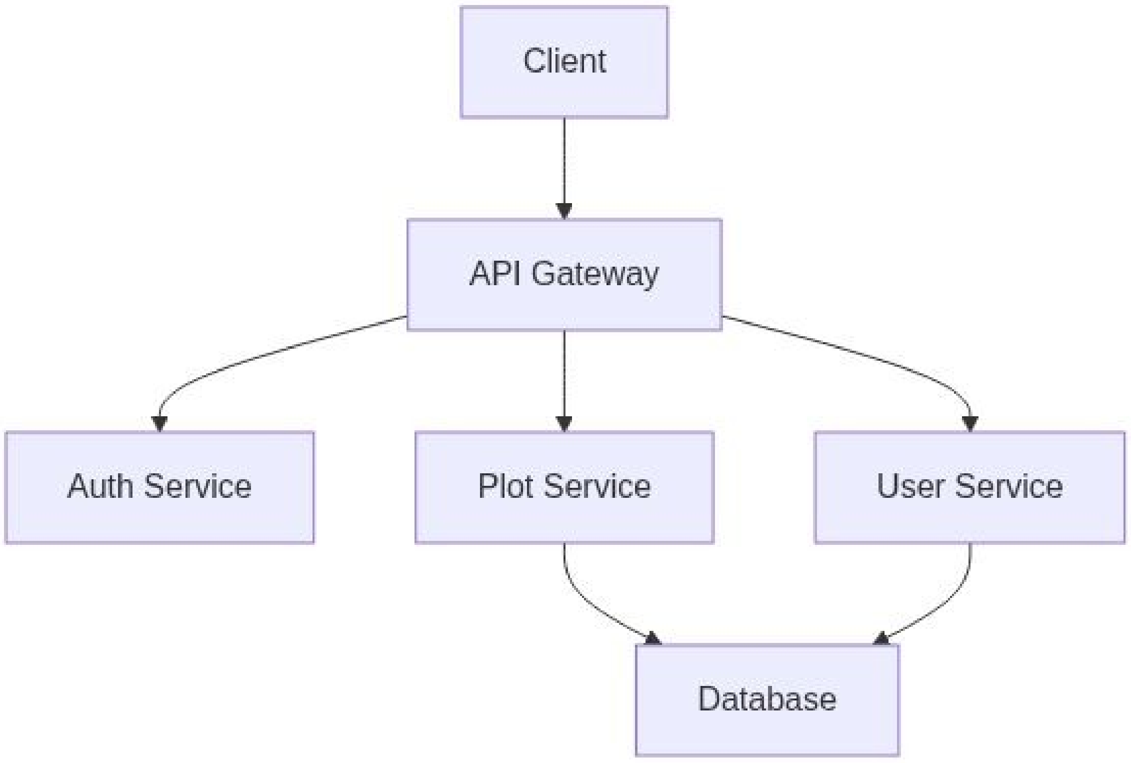
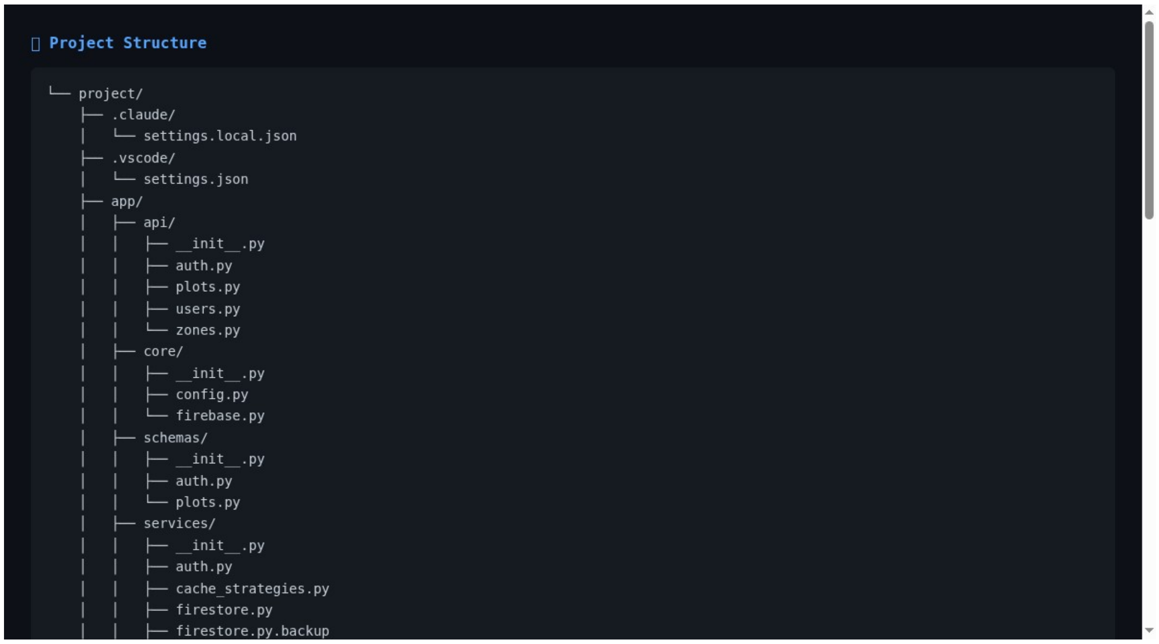


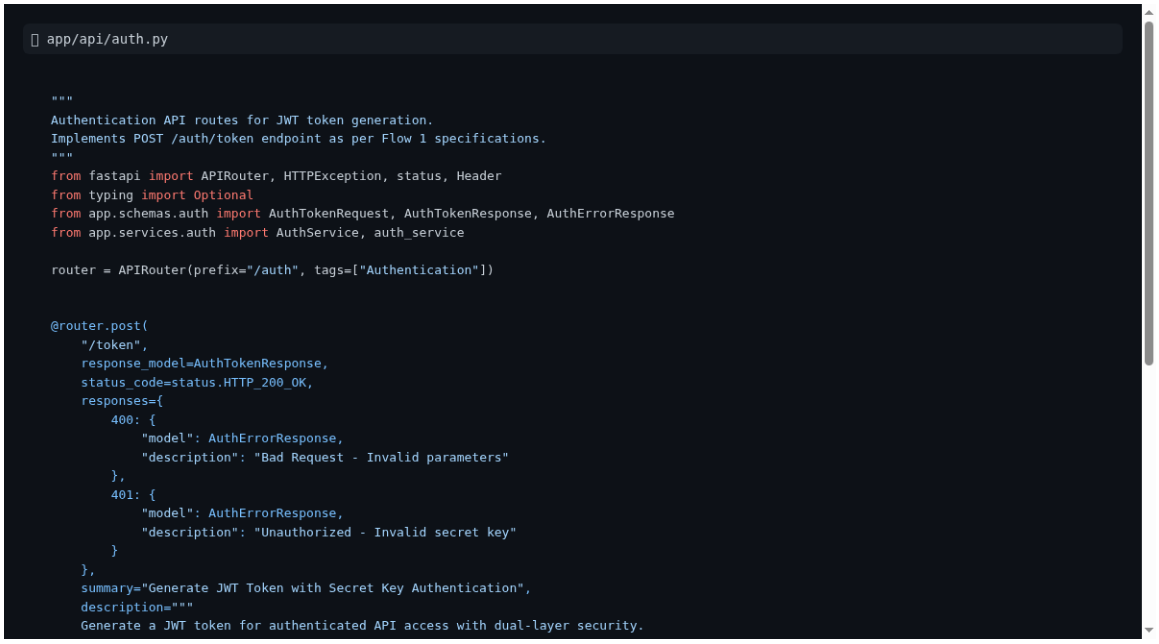
Figure: High-level system architecture diagram showing FastAPI app, database (Firestore), caching layer (Redis/In-memory), and external integrations.



Project Structure

```
└── project/
    ├── .claude/
    │   └── settings.local.json
    ├── .vscode/
    │   └── settings.json
    ├── app/
    │   ├── api/
    │   │   ├── __init__.py
    │   │   ├── auth.py
    │   │   ├── plots.py
    │   │   ├── users.py
    │   │   └── zones.py
    │   ├── core/
    │   │   ├── __init__.py
    │   │   ├── config.py
    │   │   └── firebase.py
    │   ├── schemas/
    │   │   ├── __init__.py
    │   │   ├── auth.py
    │   │   └── plots.py
    │   ├── services/
    │   │   ├── __init__.py
    │   │   ├── auth.py
    │   │   ├── cache_strategies.py
    │   │   ├── firestore.py
    │   │   ├── firestore.py.backup
```

Figure: Data flow diagram illustrating user authentication and authorization process.



```
app/api/auth.py
```

```
"""
Authentication API routes for JWT token generation.
Implements POST /auth/token endpoint as per Flow 1 specifications.
"""
from fastapi import APIRouter, HTTPException, status, Header
from typing import Optional
from app.schemas.auth import AuthTokenRequest, AuthTokenResponse, AuthErrorResponse
from app.services.auth import AuthService, auth_service

router = APIRouter(prefix="/auth", tags=["Authentication"])


@router.post(
    "/token",
    response_model=AuthTokenResponse,
    status_code=status.HTTP_200_OK,
    responses={
        400: {
            "model": AuthErrorResponse,
            "description": "Bad Request - Invalid parameters"
        },
        401: {
            "model": AuthErrorResponse,
            "description": "Unauthorized - Invalid secret key"
        }
    },
    summary="Generate JWT Token with Secret Key Authentication",
    description="""
    Generate a JWT token for authenticated API access with dual-layer security.
```

## 4. Core Components

## 4. Core Components

The Arise Plot Management API is built upon a modular architecture designed for robustness, scalability, and maintainability. This section outlines the primary components that form the backbone of the system, enabling its core functionalities.

### 4.1. API Layer

The API Layer is the primary interface through which external systems and users interact with the Arise Plot Management API. It is built using FastAPI, a modern, high-performance Python web framework, chosen for its speed, ease of use, and automatic interactive API documentation generation. This layer is responsible for:

Request Handling: Accepting incoming HTTP requests and routing them to the appropriate backend services.
Data Validation: Ensuring that incoming data conforms to predefined schemas using Pydantic models, preventing invalid data from entering the system.
Authentication & Authorization: Intercepting requests to verify user credentials and permissions, ensuring that only authorized actions are

performed based on roles and zone assignments.
Response Generation: Formatting and returning API responses in a consistent and standardized format (typically JSON).

The API Layer provides a clean and well-defined contract for all interactions with the Arise system, abstracting the underlying business logic.

### 4.2. Service Layer

The Service Layer encapsulates the core business logic of the Arise Plot Management API. It acts as an intermediary between the API Layer and the Data Layer, orchestrating complex operations and enforcing business rules. Key responsibilities include:

Business Logic Execution: Implementing the detailed workflows for managing plots, zones, and users. This includes operations like querying available plots based on various criteria, updating plot statuses, creating new zones, and managing user roles.
Data Orchestration: Coordinating operations across different data sources or internal modules if required.
Error Handling & Transformation: Managing application-specific errors and transforming them into user-friendly responses.
Decoupling: Separating the API presentation from the underlying business processes, allowing for easier updates and modifications to either layer without impacting the other.

This modular design promotes code reusability and maintainability, making it easier to extend the API's functionality.

### 4.3. Data Persistence Layer (Firebase Firestore)

The Data Persistence Layer is responsible for the secure and scalable storage and retrieval of all application data. For this project, Firebase Firestore is utilized as the primary database. Firestore is a flexible, scalable NoSQL cloud database that stores data in documents organized into collections. Its key benefits for this application include:

Scalability and Performance: Firestore automatically scales to handle massive datasets and high request volumes, essential for a multi-country plot management system.
Real-time Synchronization: Offers real-time data updates, which can be leveraged for dynamic dashboards or synchronized views if needed in future iterations.
Flexible Data Model: Its document-based structure allows for adaptable schemas, accommodating diverse plot attributes and zone information.

Managed Service: As a fully managed service by Google, it reduces the operational overhead associated with database management.

The application interacts with Firestore through dedicated utility modules (`app/core/firebase.py` and `app/services/firestore.py`), ensuring a consistent and efficient data access pattern.

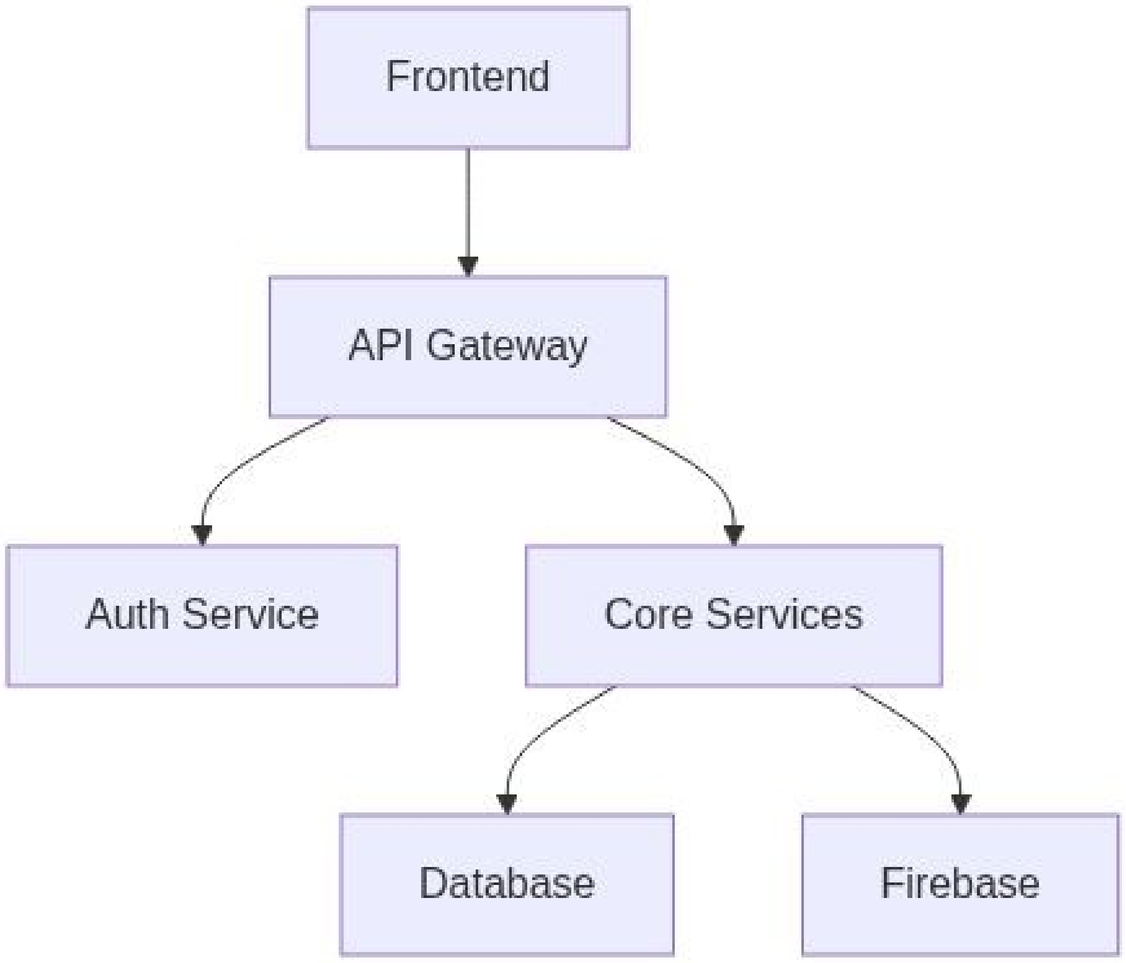
### 4.4. Authentication and Authorization Module

Security is paramount for the Arise Plot Management API. The Authentication and Authorization Module ensures that only legitimate users with appropriate permissions can access and manipulate data. This module handles:

JWT Token Generation: Upon successful authentication (based on `userId`, `role`, and `zone`), the API generates JSON Web Tokens (JWTs) containing user identity and permissions.
Token Validation: All subsequent API requests include a JWT, which is validated by the API Layer to confirm the user's identity and the token's integrity.
Role-Based Access Control (RBAC): The module enforces granular permissions based on user roles (e.g., `zone_admin`, `super_admin`) and their assigned geographical zones. This ensures that users can only perform actions within their designated scope, preventing unauthorized access or data modifications.

This robust security framework is critical for maintaining data integrity and protecting sensitive plot and user information.

Figure: Architecture Diagram: 4. Core Components



### 4.1. API Layer (FastAPI)

## 4.1. API Layer (FastAPI)

The API Layer serves as the primary gateway for all external interactions with the Arise Plot Management system. Built using **FastAPI** (https://fastapi.tiangolo.com/), a modern, high-performance Python web framework, this layer is engineered for speed, developer efficiency, and robust API development.

Key Responsibilities:

Request Handling & Routing: It receives all incoming HTTP requests, validates their structure and data against predefined schemas (using Pydantic models for data integrity), and routes them to the appropriate backend services.
Authentication & Authorization: The API Layer is responsible for enforcing security. It validates JSON Web Tokens (JWTs) provided in

requests to authenticate users and checks their assigned roles and zones to authorize access to specific resources and operations.

Response Generation: It formats and returns consistent, standardized JSON responses to clients, abstracting the underlying business logic and data operations.

Interactive Documentation: FastAPI automatically generates interactive API documentation, accessible at the `/docs` endpoint, which is crucial for developers and stakeholders to understand and test the API's capabilities.

Business Value:

This layer provides a clean, secure, and efficient interface for all system functionalities. Its design ensures that the API is easy to consume, maintain, and scale, enabling seamless integration with front-end applications and other services. The automatic documentation significantly reduces the learning curve and accelerates development cycles.

## 4.2. Authentication & Authorization

## 4.2. Authentication & Authorization

The Arise Plot Management API employs a robust security framework to ensure that only authorized users can access and manipulate data. This is achieved through a combination of authentication and role-based access control (RBAC).

Authentication:
Users are authenticated using a token-based system. Upon initial login with valid credentials (userId, role, and zone), the system generates a JSON Web Token (JWT). This token serves as a credential for all subsequent API requests. The JWT contains essential user information, including their assigned role and the geographical zone(s) they have access to.

Authorization:
The API Layer intercepts every incoming request and validates the provided JWT. This validation confirms the token's authenticity and integrity. Crucially, the system then checks the user's role and assigned zone against the requested resource or operation. This ensures that:

Role Enforcement: Users can only perform actions permitted by their specific role (e.g., `zone_admin`, `super_admin`).

Zone Restriction: Users are restricted to accessing and modifying data only within their designated geographical zones. For example, a `zone_admin` for "Benin" cannot access or alter data related to "Togo."

This layered security approach guarantees data integrity, confidentiality, and compliance with operational policies by strictly controlling access to sensitive plot and zone information.

## 4.3. Data Management & Services

## 4.3. Data Management & Services

This section details the core components responsible for managing and delivering the Arise Plot Management API's functionalities. It outlines the architecture for handling data, business logic, and security, ensuring a robust and scalable platform.

### 4.3.1. API Layer (FastAPI)

The API Layer acts as the primary interface for all external interactions with the Arise Plot Management system. Built using **FastAPI** (https://fastapi.tiangolo.com/), a modern, high-performance Python web framework, this layer is engineered for speed, developer efficiency, and robust API development.

Key Responsibilities:

Request Handling & Routing: Receives and validates incoming HTTP requests against predefined schemas, then routes them to appropriate backend services.
Authentication & Authorization: Enforces security by validating JSON Web Tokens (JWTs) and checking user roles and assigned zones for access control.
Response Generation: Formats and returns consistent JSON responses, abstracting underlying logic.
Interactive Documentation: Automatically generates interactive API documentation at the `/docs` endpoint, simplifying API understanding and testing.

Business Value:

This layer provides a secure, efficient, and easily consumable interface for all system functionalities. Its design ensures seamless integration with front-end applications and other services, while the automatic documentation accelerates development and adoption.

### 4.3.2. Service Layer

The Service Layer encapsulates the core business logic of the Arise Plot Management API, acting as an intermediary between the API Layer and the Data Persistence Layer. It orchestrates complex operations and enforces business rules, ensuring a modular and maintainable system.

Key Responsibilities:

Business Logic Execution: Implements detailed workflows for managing plots, zones, and users, including querying data, updating statuses, and creating new entities.
Data Orchestration: Coordinates operations across different data sources or internal modules.
Error Handling: Manages application-specific errors and transforms them into user-friendly responses.
Decoupling: Separates API presentation from business processes, allowing independent updates and modifications.

Business Value:

This layer promotes code reusability and maintainability by clearly defining business processes. It allows for easier extension of API functionality and ensures that the core logic is well-organized and testable.

### 4.3.3. Data Persistence Layer (Firebase Firestore)

The Data Persistence Layer is responsible for the secure and scalable storage and retrieval of all application data, utilizing **Firebase Firestore** (https://firebase.google.com/docs/firestore) as the primary NoSQL cloud database. Firestore stores data in documents organized into collections, offering significant advantages for this application.

Key Benefits:

Scalability and Performance: Automatically scales to handle large datasets and high request volumes, crucial for a multi-country plot management system.
Flexible Data Model: Its document-based structure accommodates adaptable schemas for diverse plot attributes and zone information.
Managed Service: As a fully managed service, it reduces operational overhead.

The application interacts with Firestore through dedicated utility modules, ensuring a consistent and efficient data access pattern.

Business Value:

Firestore provides a reliable, scalable, and flexible foundation for storing and managing critical plot and zone data. Its managed nature allows the development team to focus on application features rather than database administration.

### 4.3.4. Authentication and Authorization Module

The Arise Plot Management API employs a robust security framework to ensure data integrity and confidentiality. This module handles:

JWT Token Generation: Upon successful authentication, JSON Web Tokens (JWTs) containing user identity, role, and assigned zones are generated.
Token Validation: All incoming API requests include a JWT that is validated to confirm authenticity and integrity.
Role-Based Access Control (RBAC): Granular permissions are enforced based on user roles (e.g., `zone_admin`, `super_admin`) and their assigned geographical zones, restricting access and modifications to authorized scopes.

Business Value:

This layered security approach guarantees data integrity, confidentiality, and compliance with operational policies by strictly controlling access to sensitive plot and user information. It ensures that users can only perform actions within their designated scope.

### 4.4. Caching Strategies

## 4.4. Caching Strategies

To optimize application performance and reduce latency, the Arise Plot Management API employs caching mechanisms. This strategy is designed to store frequently accessed data in a readily available location, minimizing the need for repeated retrieval from the primary data source (Firebase Firestore).

Core Objective: Enhance response times and improve user experience by serving cached data for repetitive queries.

Key Benefits:

Reduced Latency: Faster retrieval of frequently accessed plot and zone information.
Improved Scalability: Offloads read operations from the primary database, allowing it to handle write-intensive tasks more efficiently.
Cost Optimization: Potentially reduces database read costs by serving data from a faster, more cost-effective cache.

Implementation Approach:

The strategy leverages an in-memory caching solution integrated within the application's service layer. This approach is suitable for data that does not change frequently or where a slight degree of staleness is acceptable. Specific data points targeted for caching include:

Frequently queried plot availability data.
Static zone configuration details.
User role and permission mappings.

The caching mechanism is managed by the `cache_strategies.py` module within the `app/services` directory. This ensures a centralized and maintainable approach to cache management, including strategies for cache invalidation when underlying data is updated to maintain data consistency.

Business Value:

By implementing effective caching, the Arise Plot Management API delivers a more responsive and efficient user experience. This is critical for applications that rely on quick data retrieval for operational efficiency and stakeholder decision-making. It also contributes to the overall robustness and scalability of the system by optimizing resource utilization.

# 5. Installation & Setup

## 5. Installation & Setup

This section outlines the essential steps and considerations for setting up and deploying the Arise Plot Management API. It is designed for both business stakeholders who need to understand the deployment requirements and technical managers responsible for the infrastructure.

### 5.1. Deployment Environment

The Arise Plot Management API is designed for deployment in a cloud-based environment, leveraging containerization for portability and scalability.

Recommended Deployment Target: Azure App Service. This Platform-as-a-Service (PaaS) offering provides a fully managed environment for hosting web applications and APIs, significantly reducing operational overhead. Azure App Service offers features such as automatic scaling, integrated monitoring, and built-in security, making it ideal for production deployments.

Containerization: The application is packaged as a Docker container. This ensures consistency across development, testing, and production environments, mitigating "it works on my machine" issues. The Docker image for the application is available in a container registry, ready for deployment.

Database: The application relies on Firebase Firestore as its primary NoSQL database. This cloud-native service provides a scalable, highly available, and flexible data storage solution. Network connectivity from the deployment environment to Firebase Firestore is a critical requirement.

Key Considerations:

Network Access: Ensure outbound network access from the deployment environment to Firebase Firestore and any other required external services.
Environment Variables: Specific environment variables are required for the application to configure its connection to Firebase, manage security secrets (JWT secret key, authentication secret key), and define operational parameters (e.g., `WEBSITE_PORT`). These must be securely configured in the deployment platform.
Resource Allocation: The chosen Azure App Service tier should provide sufficient CPU, memory, and storage to meet the application's performance and scalability needs.

### 5.2. Local Development Setup

For local development and testing, setting up a local environment that mirrors the production setup is recommended.

Prerequisites:

Docker: Install Docker Desktop (for Windows/macOS) or Docker Engine (for Linux). Docker is essential for running the application container locally.
Python: Install Python 3.11 or later. The application is developed using modern Python features.
IDE/Editor: A code editor or Integrated Development Environment (IDE) such as VS Code, PyCharm, or similar, with Python and Docker extensions, is recommended for efficient development.

Setup Steps:

1. **Clone Repository:** Obtain the project source code by cloning the repository.
2. **Install Dependencies:** Navigate to the project root and install the required Python packages using pip:
   ```bash
   pip install -r requirements.txt
   ```
   (Refer to `requirements.txt` for the definitive list of dependencies.)
3. **Configure Environment Variables:** Create a `.env` file in the project root and populate it with necessary configuration, including Firebase credentials and JWT secrets. For detailed information on required variables, consult the `app/core/config.py` file.
4. **Run Docker Compose:** Utilize Docker Compose to spin up the

application and any necessary supporting services (e.g., a local Redis instance if caching is enabled locally):

```bash
docker-compose up
```

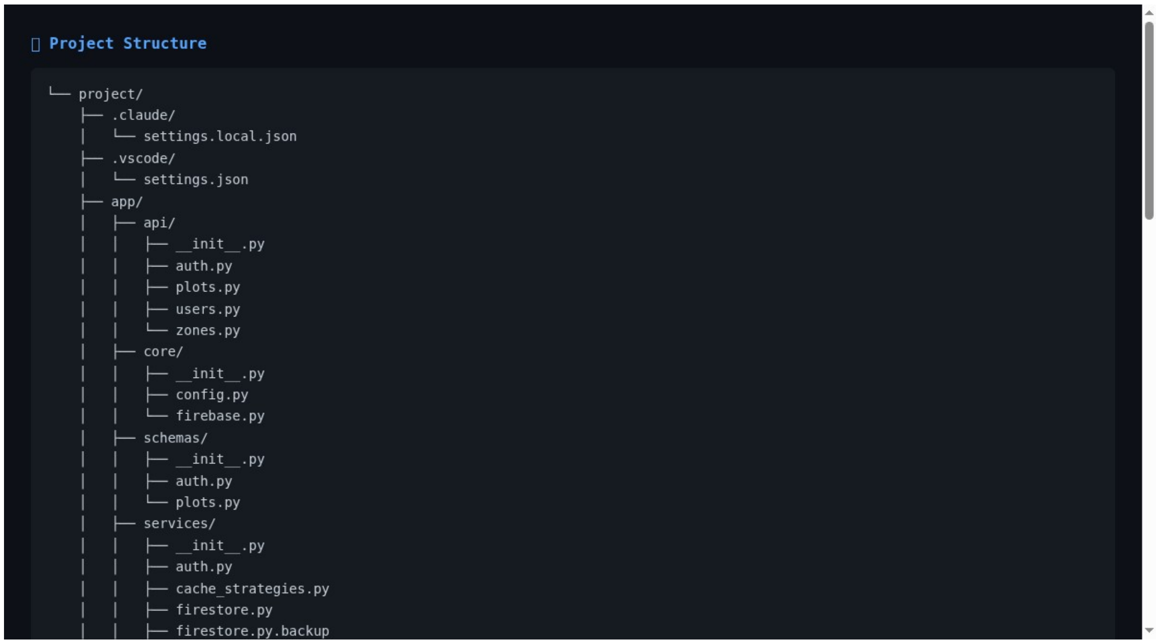
This command will build the Docker image (if not already built) and start the application container.

Verification:

Access the API documentation at `http://localhost:8000/docs` to verify the application is running.
Perform basic API calls using tools like `curl` or Postman to confirm functionality.

This local setup allows developers to iterate quickly and test changes before deploying to staging or production environments.

Figure: 5. Installation & Setup overview



Project Structure

```
└─ project/
    ├─ .claude/
    │   └─ settings.local.json
    ├─ .vscode/
    │   └─ settings.json
    ├─ app/
    │   ├─ api/
    │   │   ├─ __init__.py
    │   │   ├─ auth.py
    │   │   ├─ plots.py
    │   │   ├─ users.py
    │   │   └─ zones.py
    │   ├─ core/
    │   │   ├─ __init__.py
    │   │   ├─ config.py
    │   │   └─ firebase.py
    │   ├─ schemas/
    │   │   ├─ __init__.py
    │   │   ├─ auth.py
    │   │   └─ plots.py
    │   ├─ services/
    │   │   ├─ __init__.py
    │   │   ├─ auth.py
    │   │   ├─ cache_strategies.py
    │   │   ├─ firestore.py
    │   │   ├─ firestore.py.backup
```

## 6. Configuration

# 6. Configuration

This section details the configuration aspects of the Arise Plot Management API. It outlines how the application is configured for different environments, including local development and cloud deployments, and highlights key configuration parameters that are critical for operational success.

## 6.1. Environment Configuration

The Arise Plot Management API is designed to be highly configurable, allowing for seamless adaptation to various deployment environments. This is primarily achieved through the use of environment variables and configuration files.

Key Configuration Principles:

Separation of Concerns: Configuration is externalized from the application code, promoting flexibility and security. Sensitive information, such as API keys and secrets, is managed separately from the codebase.
Environment-Specific Settings: Different configurations can be applied for development, testing (UAT), and production environments. This ensures that the application behaves predictably and securely across its lifecycle.
Centralized Management: For cloud deployments, configuration parameters are managed through the platform's settings (e.g., Azure App Service application settings). This allows for centralized control and updates without code modifications.

Primary Configuration Mechanisms:

Environment Variables: The application reads critical configuration values from environment variables. This is a standard practice for cloud-native applications and containerized deployments. Key variables include database connection details, API secrets, and logging levels.
`.env` File (Local Development): During local development, a `.env` file is used to define environment variables. This file should not be committed to version control for security reasons.
Firebase Firestore Configuration: Connection details and project identifiers for Firebase Firestore are managed via environment variables. This ensures that the application can connect to the correct database instance for the deployed environment.
JWT Secret Key: A secure secret key is essential for signing and verifying JSON Web Tokens (JWTs). This key must be kept confidential and configured securely for each environment.

Business Value:

Robust configuration management ensures that the application can be deployed reliably and securely across different environments. It allows for easy adaptation to infrastructure changes and facilitates secure handling of sensitive credentials, contributing to the overall stability and security posture of the system.

## 6.2. Security Configuration

Security is a paramount concern for the Arise Plot Management API. Configuration plays a vital role in establishing and maintaining a secure operational environment.

Key Security Configuration Aspects:

Authentication Secrets: The API relies on a `Secret` header for initial authentication to obtain JWT tokens. This secret must be managed securely and is configured via environment variables.
JWT Secret Key: As mentioned in Environment Configuration, a strong, randomly generated secret key is used for signing JWTs. This key is critical for ensuring the integrity and authenticity of user sessions. It should be stored securely, ideally in a dedicated secrets management service (e.g., Azure Key Vault) for production environments.
Role-Based Access Control (RBAC): While the core RBAC logic resides in the application code, its effectiveness is dependent on the correct configuration of user roles and associated permissions during user creation and management. This ensures that users can only access resources and perform actions permitted by their assigned roles and zones.
HTTPS Enforcement: For cloud deployments, it is crucial to enforce HTTPS to encrypt all communication between clients and the API. This is typically configured at the platform level (e.g., Azure App Service) and may involve integrating with services like Azure CDN for SSL termination.
Rate Limiting: Although not explicitly detailed in the provided files, implementing rate limiting through API Gateway services or within the application itself is a critical security configuration for preventing abuse and denial-of-service attacks.

Business Value:

Secure configuration practices are essential for protecting sensitive plot and user data, maintaining compliance, and building trust with stakeholders. By securely managing authentication secrets and JWT keys, and by enforcing access controls, the API safeguards against unauthorized access and data breaches. This directly supports business objectives by ensuring data integrity and operational reliability.

# 7. API Reference

# 7. API Reference

This section provides a comprehensive overview of the Arise Plot Management API, detailing its endpoints, authentication mechanisms, and key functionalities. It is designed for business stakeholders and technical managers to understand the API's capabilities and how to interact with it.

## 7.1. API Overview

The Arise Plot Management API serves as the backend interface for managing land plots, zones, and user access within the Arise project. It is built using Python and the FastAPI framework, known for its high performance and developer productivity. The API exposes a set of RESTful endpoints that allow for the creation, retrieval, updating, and deletion of plot and zone data.

Key Features:

Plot Management: Enables detailed tracking and management of individual land plots, including their status, allocation, and associated business information.
Zone Management: Facilitates the definition and management of economic zones, including their geographical scope, land area, and descriptive details.
User and Role Management: Supports the creation and management of users with distinct roles (e.g., `zone_admin`, `super_admin`), enforcing role-based access control (RBAC) to ensure data security and integrity.
Authentication: Implements a secure authentication flow using JWT (JSON Web Tokens) to authorize API requests.

The API is deployed as a Docker container and is accessible via a defined base URL. For detailed endpoint specifications, request/response formats, and example usage, please refer to the [API Requests Guide] (API_REQUESTS_GUIDE.md).

Business Value:

This API provides a standardized and programmatic way to interact with the Arise Plot Management system. It enables seamless integration with front-end applications, data analysis tools, and other systems, driving operational efficiency and informed decision-making through structured access to critical land and zone data.

## 7.2. Authentication and Authorization

Secure access to the Arise Plot Management API is managed through a robust authentication and authorization system, ensuring that only authorized users can access and manipulate data.

Authentication Flow:

1. **Token Generation:** Clients must first obtain an `access_token` by making a `POST` request to the `/api/v1/auth/token` endpoint. This request requires a specific `Authorization` header containing a secret key and a JSON payload identifying the user (`userId`), their `role`, and the `zone` they are associated with.
2. **JWT Usage:** The generated `access_token` is a JSON Web Token (JWT) that is valid for a specified duration (e.g., 24 hours). This token must be included in the `Authorization: Bearer <access_token>` header for all subsequent API requests that require authentication.

Authorization:

The API enforces role-based access control (RBAC) to govern what actions users can perform and which data they can access. This is primarily determined by the `role` and `zone` information embedded within the authenticated JWT.

Zone Administrators: Users with the `zone_admin` role can typically manage plots and users only within their assigned `zone`.
Super Administrators: The `super_admin` role possesses broader privileges, allowing for system-wide management of users and potentially all zones.

This layered security approach ensures that data access is strictly controlled, adhering to the principle of least privilege and enhancing the overall security posture of the Arise Plot Management system. For detailed information on required secrets and token formats, please consult the [API Requests Guide](API_REQUESTS_GUIDE.md).

# 8. Deployment Guide

# 8. Deployment Guide

This section outlines the essential steps and considerations for deploying the Arise Plot Management API. It provides a high-level overview of the deployment process, target environments, and key configuration requirements.

## 8.1. Deployment Overview

The Arise Plot Management API is designed for deployment as a Docker container on cloud platforms, specifically targeting Azure App Service. This approach ensures scalability, reliability, and ease of management. The deployment process involves containerizing the application, configuring the necessary cloud infrastructure, and setting up continuous integration and deployment (CI/CD) pipelines for efficient updates.

Key Deployment Stages:

1. **Containerization:** The application is packaged into a Docker image, ensuring a consistent runtime environment across development and production.
2. **Infrastructure Setup:** Cloud resources, such as Azure App Service, Azure Container Registry, and relevant networking components, are provisioned and configured.
3. **Configuration Management:** Environment-specific settings, including database credentials, API secrets, and logging configurations, are securely managed.
4. **Deployment Execution:** The Docker image is deployed to the target Azure App Service instance.
5. **Monitoring and Maintenance:** Post-deployment, continuous monitoring is established to ensure application health, performance, and security. Regular updates and maintenance are performed via CI/CD pipelines.

Business Benefits:

Scalability: Cloud-native deployment allows the API to scale automatically based on demand, ensuring consistent performance during peak loads.
Reliability: Managed services like Azure App Service provide high availability and robust infrastructure, minimizing downtime.
Efficiency: Containerization and CI/CD streamline the deployment process, enabling faster iteration cycles and quicker delivery of new features.
Security: Secure configuration and managed infrastructure contribute to a robust security posture, protecting sensitive data.

## 8.2. Target Deployment Environment: Azure App Service

The primary deployment target for the Arise Plot Management API is **Azure App Service**, leveraging its "Web App for Containers" feature. This Platform-as-a-Service (PaaS) offering provides a fully managed environment for hosting containerized web applications.

Why Azure App Service?

Managed Infrastructure: Azure handles the underlying infrastructure, including servers, operating systems, and patching, allowing the development team to focus on application logic.
Scalability and Performance: App Service offers robust auto-scaling

capabilities, ensuring the API can handle varying traffic loads efficiently. It provides a 99.95% availability SLA.

Integrated CI/CD: Seamless integration with Azure DevOps and other CI/CD tools facilitates automated deployments from code commits to production.

Security Features: Built-in security features, including SSL/TLS certificate management, network security configurations, and integration with Azure Key Vault for secret management, enhance the application's security.

Cost-Effectiveness: For this application's needs, the Standard S1 tier of Azure App Service offers a balanced combination of performance, features, and cost. Detailed cost analysis is available in the CLIENT-MEETING-SOLUTIONS.md document.

Key Deployment Components:

Azure Container Registry (ACR): Used to store and manage the Docker images of the Arise Plot Management API.

Azure App Service Plan: Defines the compute resources (tier, OS, scale settings) for the deployed application. The `Standard S1` tier is recommended for production.

Application Insights: Integrated for comprehensive monitoring of application performance, availability, and errors.

For a detailed implementation plan and cost breakdown, please refer to the CLIENT-MEETING-SOLUTIONS.md document.

# 9. Development Guide

# 9. Development Guide

This section provides guidance for developers and technical managers on how to understand, interact with, and contribute to the Arise Plot Management API project. It outlines the project's structure, key development practices, and essential tools.

## 9.1. Project Overview and Structure

The Arise Plot Management API is a Python-based FastAPI application designed for managing land plots and zones. It is containerized using Docker for deployment on cloud platforms like Azure App Service. The project is organized to facilitate maintainability and scalability.

Key Directories and Their Purpose:

`app/`: Contains the core application logic, including:
`api/`: Defines the API endpoints and request/response models.
`core/`: Houses essential configurations, database connections, and utility functions.
`schemas/`: Defines data structures for request and response payloads.
`services/`: Implements business logic and interactions with external services like Firebase.
`utils/`: Contains general utility functions.
`uat-distribution/`: May contain deployment-specific artifacts or configurations for User Acceptance Testing (UAT) environments.
Root Directory: Contains configuration files (`.gitignore`, `Dockerfile`, `docker-compose.yml`), documentation (`README.md`, `API_REQUESTS_GUIDE.md`), and dependency management files (`requirements.txt`).

Development Philosophy:

The project emphasizes clear code structure, modularity, and adherence to best practices for building robust and maintainable APIs. This includes using FastAPI for rapid API development, Docker for consistent environments, and standard Python packaging for dependency management.

Business Value:

A well-structured project facilitates faster development cycles, easier onboarding of new developers, and reduced technical debt. This leads to more reliable software, quicker feature delivery, and a more secure and scalable application, ultimately supporting business objectives through efficient operations and data management.

## 9.2. Development Environment Setup

Establishing a consistent and functional development environment is crucial for efficient API development and troubleshooting. This guide outlines the necessary tools and configurations.

Prerequisites:

Python: Version 3.11 or higher is recommended. Installation instructions can be found on the [official Python website](https://www.python.org/downloads/).
Docker: Essential for containerizing the application and running it locally. Download and install Docker Desktop from [docker.com](https://www.docker.com/products/docker-desktop/).
Git: For version control. Available at git-scm.com.

Local Development Workflow:

1. **Clone the Repository:**
   ```bash
   git clone <repository_url>
   cd arise_fastapi
   ```
2. **Install Dependencies:**
   Create a virtual environment (recommended) and install project dependencies:
   ```bash
   python -m venv venv
   source venv/bin/activate  # On Windows use `venv\Scripts\activate`
   pip install -r requirements.txt
   ```
3. **Configure Environment Variables:**
   The application relies on environment variables for configuration (e.g., database credentials, API secrets). Refer to `app/core/config.py` for required variables. A `.env` file can be created in the project root for local development.
4. **Run the Application:**
   Use the provided `start.sh` script or run the FastAPI application directly:
   ```bash
   uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
   ```
   Alternatively, build and run the Docker image:
   ```bash
   docker-compose up
   ```

Testing:

Unit and Integration Tests: The project structure should include a `tests/` directory for comprehensive test coverage. Running tests ensures code quality and prevents regressions. Refer to `pytest` documentation for best practices.
API Interaction: Use tools like Postman or `curl` to interact with the API endpoints during development. The `API_REQUESTS_GUIDE.md` file provides detailed examples.

Business Value:

A streamlined development environment setup reduces the time and effort required for developers to become productive. This accelerates feature development, bug fixing, and overall project velocity. Consistent local environments also minimize "it works on my machine" issues, leading to more reliable deployments.

## 9.3. API Interaction and Documentation

Understanding how to interact with the Arise Plot Management API is crucial for both developers and stakeholders. This section highlights the primary resources for API interaction.

API Base URL and Endpoints:

The API is accessible via a base URL, which may vary depending on the deployment environment (e.g., development, UAT, production). Refer to environment-specific documentation for the exact URL. The API is versioned, typically accessed via `/api/v1/`.

Authentication:

The API employs a secure JWT-based authentication system.

1. **Token Acquisition:** Obtain an access token by POSTing to the `/api/v1/auth/token` endpoint with valid credentials. This requires a specific `Authorization` header containing a secret key.
2. **Authenticated Requests:** Include the obtained JWT in the `Authorization: Bearer <token>` header for all subsequent requests to protected endpoints.

Key Documentation Resources:

`API_REQUESTS_GUIDE.md`: This document is the primary resource for understanding how to make API requests. It provides:

Detailed examples of `curl` commands for each endpoint.

Request body structures (JSON).

Authentication procedures.

Sample responses.

Information on error handling and common status codes.

A comprehensive list of available endpoints for plot management, zone management, and user management.

[Link to API Requests Guide](API_REQUESTS_GUIDE.md)

Interactive API Documentation (`/docs`): The FastAPI application automatically generates interactive API documentation (Swagger UI) accessible at the `/docs` endpoint of the running API. This provides an intuitive interface for exploring endpoints, parameters, and making test requests directly from the browser.

Business Value:

Clear and accessible API documentation empowers developers to integrate with the system efficiently and enables business stakeholders to understand the API's capabilities without needing to dive into the codebase. This promotes interoperability, reduces integration time, and ensures consistent usage of the API, supporting broader business goals.

## 9.4. Logging and Monitoring

Effective logging and monitoring are essential for maintaining the health, performance, and security of the Arise Plot Management API, especially in production environments.

Logging Strategy:

Application Logs: The application should log critical events, errors, and significant operational activities. This includes:
Startup and shutdown events.
Incoming API requests and their outcomes (success/failure).
Authentication attempts (successful and failed).
Errors encountered during processing, including tracebacks.
Interactions with external services (e.g., Firebase).
Log Levels: Utilize standard logging levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize log messages.
Centralized Logging: For production deployments, logs should be aggregated into a centralized logging system (e.g., Azure Application Insights, Elasticsearch, Splunk) for efficient analysis and long-term storage.

Monitoring:

Health Checks: Implement a dedicated health check endpoint (e.g., `/health`) that provides a quick status of the application's operational readiness.
Performance Metrics: Monitor key performance indicators (KPIs) such as response times, request throughput, and error rates.
Resource Utilization: Track CPU, memory, and network usage of the application instances.
Alerting: Configure alerts for critical events, such as high error rates, performance degradation, or resource exhaustion. Alerts should be routed to the appropriate support teams.

Tools and Services:

Python `logging` module: The standard library for application-level logging.
Azure Application Insights: Integrated with Azure App Service for comprehensive application performance monitoring (APM), log aggregation, and alerting.
Docker Logs: Container logs can be accessed via `docker logs <container_id>` or managed by the container orchestration platform.

Business Value:

Robust logging and monitoring provide real-time insights into the application's behavior. This enables proactive identification and resolution of issues, minimizes downtime, ensures optimal performance, and aids in security incident investigation. Ultimately, it contributes to a reliable and trustworthy service delivery, supporting business continuity and customer satisfaction.

## 9.5. Security Best Practices

Security is a critical consideration throughout the development lifecycle of the Arise Plot Management API. Adhering to best practices ensures the protection of sensitive data and the integrity of the system.

Key Security Measures:

Secure Authentication:

Secret Management: The initial authentication secret must be kept confidential and managed securely, ideally through environment variables or a dedicated secrets management service (e.g., Azure Key Vault) in production.

JWT Security: Use strong, randomly generated secret keys for signing JWTs. Ensure tokens have appropriate expiration times. Avoid embedding sensitive information directly in the JWT payload.

Authorization (RBAC):

Implement strict role-based access control to ensure users can only access resources and perform actions permitted by their assigned roles and zones.

Validate user permissions on every sensitive API request.

Input Validation:

Sanitize and validate all user inputs to prevent common vulnerabilities such as SQL injection and cross-site scripting (XSS). FastAPI's Pydantic models aid significantly in this.

HTTPS Enforcement:

Always use HTTPS for all API communication, especially in production, to encrypt data in transit. This is typically configured at the deployment platform level (e.g., Azure App Service).

Dependency Management:

Regularly update dependencies to patch known vulnerabilities. Use tools like `pip-audit` or GitHub's Dependabot for vulnerability scanning.

Secure Configuration:

Avoid hardcoding sensitive information (API keys, passwords) directly in the code. Use environment variables or secure configuration management tools.

Error Handling:
Do not expose detailed error messages or stack traces to the client in production environments, as this can reveal system internals. Log detailed errors server-side.

Business Value:

Implementing strong security measures protects sensitive business data (plot information, user details) from unauthorized access and breaches. This builds trust with users and stakeholders, ensures compliance with data protection regulations, and prevents costly security incidents, thereby safeguarding the business's reputation and operational stability.

## 9.6. Code Contribution Guidelines

To ensure a collaborative and efficient development process, adherence to coding standards and contribution guidelines is essential.

General Principles:

Readability: Write clean, well-commented, and easily understandable code. Follow standard Python style guides (PEP 8).
Modularity: Design components to be reusable and independent. Separate concerns into distinct modules and classes.
Testability: Write code that is easy to test. Ensure comprehensive unit and integration tests are provided for all new features and bug fixes.
Documentation: Keep code documentation up-to-date. This includes docstrings for functions and classes, and updates to relevant Markdown files when significant changes are made.

Contribution Workflow:

1. **Fork the Repository:** Create a personal fork of the main repository.
2. **Create a Branch:** Create a new feature branch for your changes (e.g., `feature/add-new-plot-endpoint`).
3. **Develop and Test:** Implement your changes and write corresponding tests. Ensure all tests pass.
4. **Commit Changes:** Write clear and concise commit messages that explain the purpose of the changes.
5. **Pull Request (PR):** Submit a pull request from your branch to the main repository.
   Clearly describe the changes and their purpose.
   Reference any related issues or tickets.
6. **Code Review:** Participate actively in code reviews, providing constructive feedback and addressing comments on your PR.
7. **Merge:** Once approved, the PR will be merged into the main branch.

Tooling:

Linters and Formatters: Utilize tools like `flake8`, `black`, and `isort` to maintain code quality and consistency. These can often be integrated into your IDE or run as pre-commit hooks.
Version Control: Use Git for all code management.

Business Value:

Clear contribution guidelines foster a consistent codebase, improve code quality, and streamline the review process. This leads to a more stable and maintainable application, faster delivery of features, and better collaboration among development team members, ultimately contributing to the project's long-term success.

# 10. Troubleshooting

# 10. Troubleshooting

This section provides guidance for diagnosing and resolving issues encountered with the Arise Plot Management API. It covers common problems, diagnostic tools, and strategies for effective troubleshooting.

## 10.1. Common Issues and Resolutions

Several common issues may arise during the development, deployment, or operation of the Arise Plot Management API. Understanding these potential problems and their solutions can significantly expedite issue resolution.

### Container Startup Failures (Exit Code 1)

Problem: The Docker container fails to start on the deployment platform (e.g., Azure App Service), exiting with code 1.

Root Cause: This typically indicates an error during the application's initialization or startup sequence within the container. Common causes include:
Missing Environment Variables: Critical configuration parameters required by the application are not set.
Application Errors: Uncaught exceptions during FastAPI application startup.
Dependency Issues: Required libraries or services are not available or configured correctly.
Port Conflicts: The application attempts to bind to a port that is already in use.

Resolution Strategies:
1. **Review Container Logs:** Access and analyze the detailed logs generated by the container. The `RN-APS-UAT-2001_logs.Logs` file provides an example of container lifecycle logs. Look for specific error messages indicating the cause of the failure.
2. **Verify Environment Variables:** Ensure all necessary environment variables are correctly configured in the deployment environment. Refer to `app/core/config.py` for a list of expected variables.
3. **Check Application Code:** Examine `app/main.py` and related startup scripts for any explicit error handling or configuration logic that might be failing.
4. **Test Locally:** Ensure the Docker container runs successfully on a local development environment using `docker-compose up`. This helps isolate whether the issue is environment-specific or code-related.
5. **Interactive Debugging:** If possible, use the deployment platform's tools (e.g., Azure App Service Bash) to execute commands within the running container to inspect file systems, check running processes, and manually trigger application startup.

Business Impact: Inability to start the application container directly impacts service availability and prevents users from accessing API functionalities.

### Authentication and Authorization Errors

Problem: Users are unable to authenticate or access specific API endpoints due to authorization failures.

Root Cause:
 Invalid Credentials: Incorrect `userId`, `role`, or `zone` provided during token acquisition.
 Expired or Invalid JWT: The provided JWT token is expired, tampered with, or not properly formatted.
 Insufficient Permissions: The user's role or assigned zone does not grant access to the requested resource.

Resolution Strategies:
1. **Verify Credentials:** Double-check the `userId`, `role`, and `zone` used in the `/api/v1/auth/token` request. Refer to `API_REQUESTS_GUIDE.md` for correct parameters.
2. **Check Token Validity:** Ensure the JWT is correctly included in the `Authorization: Bearer <token>` header and has not expired.
3. **Review Role-Based Access Control (RBAC):** Consult the application's logic (e.g., in `app/services/auth.py`) and documentation to confirm the user's role and zone permissions align with the requested action.

4. **Examine API Response Codes:** Pay attention to HTTP status codes (e.g., 401 Unauthorized, 403 Forbidden) and the accompanying error messages, which provide clues about the specific failure. Refer to the error handling section in `API_REQUESTS_GUIDE.md`.

Business Impact: Prevents legitimate users from performing necessary actions, disrupting workflows and potentially leading to lost productivity or business opportunities.

### Data Retrieval or Update Failures

Problem: API requests to retrieve or update plot, zone, or user data are failing.

Root Cause:
Incorrect Parameters: Missing or malformed query parameters or request body fields.
Data Integrity Issues: Inconsistent or invalid data in the backend database (e.g., Firestore).
Firebase Connectivity Problems: Issues connecting to or interacting with the Firebase backend.
API Logic Errors: Bugs in the service layer or data handling logic.

Resolution Strategies:
1. **Validate Request Parameters:** Ensure all required parameters (`country`, `zoneCode`, `phase`, `plotName`, etc.) are correctly provided as per `API_REQUESTS_GUIDE.md`.
2. **Inspect Database Records:** If possible, verify the existence and correctness of the target data in Firebase.
3. **Check Firebase Connection:** Ensure the application can successfully connect to Firebase. The `/firebase-test` endpoint can be used for this.
4. **Analyze Application Logs:** Review server-side logs for errors related to database operations or Firebase interactions.
5. **Test with Sample Data:** Use the provided `Arise FastAPI Backend.postman_collection_Azure_UAT` or `test_api.sh` script to test endpoints with known good data.

Business Impact: Inaccurate or unavailable data hinders decision-making and operational efficiency. Failure to update records can lead to outdated information and operational discrepancies.

## 10.2. Diagnostic Tools and Techniques

Effective troubleshooting relies on utilizing the right tools and techniques to gather information and pinpoint the root cause of issues.

### Logging and Monitoring

Application Logs: The Python `logging` module is used to record events, errors, and operational details within the application. These logs are crucial for understanding the application's internal state and identifying errors. For production environments, these logs should be aggregated into a centralized system like Azure Application Insights.
Container Logs: When deploying via Docker, container logs provide insight into the container's lifecycle, including startup, runtime, and exit events. The `RN-APS-UAT-2001_logs.Logs` file exemplifies the type of information found in these logs.
Health Check Endpoint (`/health`): This endpoint provides a quick status check of the application's operational readiness. It's a vital first step in diagnosing service availability issues.
Firebase Connectivity Test (`/firebase-test`): This endpoint verifies the application's ability to connect to and interact with the Firebase backend, a critical dependency.

### API Testing Tools

`curl`: A command-line tool for making HTTP requests. It is extensively used in `API_REQUESTS_GUIDE.md` to demonstrate API interactions and can be used for manual testing and debugging.
Postman: A popular API development and testing platform. The `Arise FastAPI Backend.postman_collection_Azure_UAT` file provides a pre-configured collection for testing the API, including authentication and various endpoints.
Interactive API Documentation (`/docs`): The FastAPI application automatically generates interactive API documentation (Swagger UI) accessible at the `/docs` endpoint. This allows for easy exploration of endpoints, parameters, and direct testing of API calls.

### Code Analysis and Debugging

Source Code Review: Thoroughly reviewing the codebase, particularly critical files like `app/main.py`, `app/core/config.py`, and service modules (e.g., `app/services/auth.py`), is essential for understanding application logic and identifying potential error sources.
Environment Variables: Understanding and verifying the correct configuration of environment variables is paramount, as they drive application behavior and connectivity.
Local Development Environment: Replicating the deployment environment locally using Docker Compose (`docker-compose up`) allows for direct debugging and testing of the application's behavior outside of the production infrastructure.

Business Value: A systematic approach to troubleshooting, leveraging detailed logs, API testing tools, and code analysis, minimizes downtime, reduces the mean time to resolution for issues, and ensures the stability and reliability of the Arise Plot Management API. This directly supports business operations by providing a dependable service.

## 10.3. Escalation Procedures

When issues cannot be resolved through standard troubleshooting steps, an escalation process ensures that the problem is addressed by the appropriate personnel.

### Internal Escalation

1. **Level 1 Support (Development Team):** Initial troubleshooting is handled by the core development team. This involves reviewing logs, testing API endpoints, and performing code analysis.
2. **Level 2 Support (Senior Engineers/Architects):** If the issue is complex or requires deeper system knowledge, it is escalated to senior engineers or the technical architect. This may involve in-depth performance analysis, infrastructure review, or architectural decision-making.
3. **Level 3 Support (Platform/DevOps Team):** For issues related to the deployment infrastructure (e.g., Azure App Service configuration, network problems, Docker environment), escalation to the platform or DevOps team is necessary.

### External Escalation

Cloud Provider Support: For issues directly related to the cloud infrastructure (e.g., Azure App Service outages, performance degradation of Azure services), engage the respective cloud provider's support channels. Refer to Azure Support documentation for detailed procedures.
Third-Party Service Providers: If the issue involves external services (e.g., Firebase, external APIs), contact the support channels of those providers.

### Escalation Triggers

Escalation should be considered when:
The issue is causing a critical outage or significant business disruption.
Standard troubleshooting steps have been exhausted without resolution.
The issue requires expertise beyond the current team's capabilities.
A predefined Service Level Agreement (SLA) for issue resolution is at risk of being breached.

Information to Provide During Escalation:
Clear description of the problem.

Observed symptoms and error messages.
Steps already taken to resolve the issue.
Impact on business operations.
Relevant logs and diagnostic data.

Business Value: A well-defined escalation process ensures that critical issues are addressed promptly and effectively by the right resources. This minimizes business impact, maintains service availability, and demonstrates a commitment to robust operational support.