SDS Chemical Inventory System - Project Documentation

Executive Summary

This document provides comprehensive documentation for the SDS Chemical Inventory & Reporting System, a backend application built with FastAPI, SQLAlchemy ORM, Alembic, and asyncpg. The system demonstrates hybrid database access patterns, containerization with Docker, and automated deployment processes.

---

1. Setup Instructions

Quick Start (Docker - Recommended)

1. **Extract the project zip file**

1. **Navigate to project directory:**

`bash

cd sds_inventory_system

`

1. **Run the application:**

`bash

./run.sh

`

1. **Access the application:**

- API: http://localhost:8000
- Documentation: http://localhost:8000/docs
- Health Check: http://localhost:8000/health

Local Development Setup

1. **Create Python environment:**

```bash
conda env create -f environment.yml
conda activate sds_inventory
```

1. **Install dependencies:**

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

1. **Start PostgreSQL:**

```bash
docker run -d --name postgres \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=postgres \
-e POSTGRES_DB=sds_inventory \
-p 5432:5432 \
postgres:16-alpine
```

1. **Run migrations:**

`bash

alembic upgrade head

`

1. **Start application:**

`bash

uvicorn app.main:app --reload

`

Azure PostgreSQL Configuration

To use Azure PostgreSQL, update the `.env` file:

```
DATABASE_HOST=your-server.postgres.database.azure.com
DATABASE_PORT=5432
DATABASE_NAME=sds_inventory
DATABASE_USER=your-username@your-server
DATABASE_PASSWORD=your-password
ENVIRONMENT=azure
```

---

2. Architecture Overview

Technology Stack

- **Framework:** FastAPI (Python 3.13)
- **ORM:** SQLAlchemy 2.0
- **Database:** PostgreSQL 16
- **Migrations:** Alembic
- **Async Driver:** asyncpg
- **Containerization:** Docker & Docker Compose

- **Validation:** Pydantic

Hybrid Database Access Pattern

The application demonstrates two database access patterns:

1. **ORM Access (SQLAlchemy):**

- POST /chemicals/ (Create)
- GET /chemicals/ (List all)
- PUT /chemicals/{id} (Update)
- DELETE /chemicals/{id} (Delete)
- POST /chemicals/{id}/log (Create log)

1. **Direct SQL Access (asyncpg):**

- GET /chemicals/{id} (Get by ID)
- GET /chemicals/{id}/logs (Get logs)

This hybrid approach showcases:

- Complex queries optimization with raw SQL
- ORM convenience for CRUD operations
- Performance benefits of asyncpg for read-heavy operations

Project Structure

```
sds_inventory_system/
├── app/
│   ├── api/
│   │   ├── chemicals.py       # API endpoints
│   │   └── schemas.py          # Pydantic models
│   ├── core/
│   │   └── config.py           # Configuration
│   ├── db/
```

```
│   │   ├── base.py            # Database setup
│   │   └── session.py         # Session management
│   ├── models/
│   │   ├── chemical.py        # Chemical model
│   │   └── inventory_log.py   # Log model
│   └── main.py                # FastAPI app
├── alembic/                   # Migrations
├── docker-compose.yml         # Orchestration
├── Dockerfile                # Container definition
├── entrypoint.sh             # Startup script
├── run.sh                   # Automation script
└── requirements.txt         # Dependencies
```

---

3. Implementation Details

Database Models

Chemical Model:

```
- id: Integer (Primary Key)
- name: String
- cas_number: String (Unique)
- quantity: Float
- unit: String
- created_at: DateTime
- updated_at: DateTime
```

InventoryLog Model:

```
- id: Integer (Primary Key)
- chemical_id: Integer (Foreign Key)
- action_type: Enum (add/remove/update)
- quantity: Float
- timestamp: DateTime
```

API Endpoints

Method	Endpoint	Description	Access Type
POST	/chemicals/	Create chemical	ORM
GET	/chemicals/	List chemicals	ORM
GET	/chemicals/{id}	Get by ID	asyncpg
PUT	/chemicals/{id}	Update chemical	ORM
DELETE	/chemicals/{id}	Delete chemical	ORM
POST	/chemicals/{id}/log	Create log	ORM
GET	/chemicals/{id}/logs	Get logs	asyncpg

Key Features

1. **Automatic Migrations:** Alembic migrations run on container startup
1. **Environment Detection:** Automatically switches between local and Azure PostgreSQL
1. **Health Monitoring:** Built-in health check endpoint
1. **API Documentation:** Auto-generated Swagger/OpenAPI documentation
1. **Test Suite:** Comprehensive test script for all endpoints

---

4. Challenges and Solutions

Challenge 1: Hybrid Database Access Implementation

Problem: Implementing both ORM and raw SQL access patterns in the same application while maintaining clean architecture.

Solution:

- Created separate dependency injection functions for SQLAlchemy sessions and asyncpg connections
- Used FastAPI's dependency system to inject the appropriate database connection
- Maintained clear separation between ORM operations and raw SQL queries

Implementation:

```
# ORM dependency
async def get_db() -> AsyncSession:
    async with AsyncSessionLocal() as session:
        yield session

# asyncpg dependency
async def get_asyncpg_connection():
    conn = await asyncpg.connect(...)
    try:
        yield conn
    finally:
        await conn.close()
```

Challenge 2: Automatic Migration on Startup

Problem: Ensuring database migrations run automatically before the API starts, especially in containerized environments.

Solution:

- Created an entrypoint.sh script that:

1. Waits for PostgreSQL to be ready
1. Runs Alembic migrations
1. Starts the FastAPI application

- Used Docker health checks to ensure proper service dependencies

Implementation:

```
# Wait for database
while ! pg_isready -h $DATABASE_HOST; do
    sleep 2
done
# Run migrations
alembic upgrade head
# Start app
uvicorn app.main:app
```

Challenge 3: Environment Configuration Flexibility

Problem: Supporting both local Docker PostgreSQL and Azure PostgreSQL with minimal configuration changes.

Solution:

- Used environment variables with sensible defaults
- Created a Settings class using Pydantic BaseSettings
- Implemented dynamic connection string generation
- Provided clear .env examples for different environments

Challenge 4: Async/Await Pattern Consistency

Problem: Maintaining consistent async patterns across ORM and raw SQL operations.

Solution:

- Used SQLAlchemy's async engine and session
- Implemented all endpoints as async functions
- Properly handled connection lifecycle with context managers
- Ensured proper error handling for both access patterns

Challenge 5: Docker Networking

Problem: Ensuring proper communication between containers while maintaining local development compatibility.

Solution:

- Used Docker Compose service names for inter-container communication
- Configured environment variables to switch between 'localhost' and 'db' hostname
- Implemented health checks to ensure service readiness

---

5. Testing

Test Coverage

The included `test_api.py` script tests:

1. **Create Operation:** POST /chemicals/
1. **Read Operations:** GET /chemicals/, GET /chemicals/{id}
1. **Update Operation:** PUT /chemicals/{id}
1. **Delete Operation:** DELETE /chemicals/{id}
1. **Log Creation:** POST /chemicals/{id}/log
1. **Log Retrieval:** GET /chemicals/{id}/logs
1. **Error Handling:** 404 responses for non-existent resources
1. **Data Validation:** Pydantic model validation

Running Tests

```
# Ensure API is running
./run.sh

# In another terminal
python test_api.py
```

Expected Output:

```
=== Testing SDS Chemical Inventory API ===
1. Creating a new chemical...
    Chemical created with ID: 1
2. Getting all chemicals...
    Found 1 chemical(s)
3. Getting chemical by ID (using asyncpg)...
    Retrieved chemical: Ethanol
[... additional test results ...]
=== All tests completed! ===
```

---

6. Performance Considerations

Optimizations Implemented

1. **Connection Pooling:** Used SQLAlchemy's connection pool for ORM operations
1. **Async Operations:** All database operations are asynchronous
1. **Direct SQL for Reads:** Used asyncpg for read-heavy operations
1. **Indexed Fields:** Primary keys and foreign keys are indexed
1. **Pagination Support:** List endpoints support skip/limit parameters

Scalability Considerations

- **Horizontal Scaling:** Application is stateless and can be scaled horizontally
- **Database Connections:** Connection pooling prevents connection exhaustion
- **Migration Safety:** Alembic ensures database schema consistency across instances

---

7. Security Considerations

1. **Environment Variables:** Sensitive data stored in .env file, not in code

1. **SQL Injection Prevention:**

- ORM queries are parameterized by default
- asyncpg uses parameterized queries ($1, $2)

1. **Input Validation:** Pydantic models validate all input data

1. **Docker Security:** Non-root user in production containers (can be added)

---

8. Time Tracking

Task	Hours Spent
Project Setup & Environment	0.5
Database Models & Migrations	1.0
API Implementation (Hybrid)	2.0
Docker Configuration	1.0
Testing & Debugging	1.5
Documentation	1.0
Total	**7.0 hours**

---

9. Future Enhancements

Potential improvements for production deployment:

1. **Authentication & Authorization:** Add JWT-based authentication

1. **Caching:** Implement Redis for frequently accessed data

1. **Monitoring:** Add Prometheus metrics and Grafana dashboards
1. **CI/CD:** GitHub Actions for automated testing and deployment
1. **API Versioning:** Implement proper API versioning strategy
1. **Rate Limiting:** Add rate limiting to prevent abuse
1. **Audit Logging:** Comprehensive audit trail for all operations
1. **Backup Strategy:** Automated database backup procedures

---

10. Conclusion

This project successfully demonstrates:

- Full CRUD operations with FastAPI
- Hybrid database access (ORM + raw SQL)
- Automatic migration system with Alembic
- Docker containerization
- Environment configuration flexibility
- Comprehensive testing
- Production-ready code structure

Appendix A: Quick Command Reference

```
# Start application
./run.sh

# View logs
docker-compose logs -f

# Stop application
```

```
docker-compose down

# Restart application
docker-compose restart

# Stop and restart with fresh containers
docker-compose down && docker-compose up -d

# Reset database
docker-compose down -v

# Run migrations manually
alembic upgrade head

# Create new migration
alembic revision --autogenerate -m "description"

# Run tests
python test_api.py

# Access PostgreSQL
docker exec -it sds_inventory_system_db_1 psql -U postgres -d sds_inventory
```

Appendix B: Environment Variables

Variable	Description	Default	Example
DATABASE_HOST	Database hostname	localhost	db
DATABASE_PORT	Database port	5432	5432
DATABASE_NAME	Database name	sds_inventory	sds_inventory
DATABASE_USER	Database username	postgres	postgres

Variable	Description	Default	Example
DATABASE_PASSWORD	Database password	postgres	secret123
ENVIRONMENT	Environment name	local	azure

End of Documentation