Tarikhi

Tunisian Historical Sites API

Web Services Final Project

by

Roea Boubaker

Major: BA
Minor: IT

Supervisor: Dr. Montasser Ben Messoued

Tunis Business School

TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

Declaration of Authorship

I declare on my honor that the work presented in this report, titled "Tarikhi" is original and was carried out by *Roea Boubaker* under the supervision of Dr. *Montasser Ben Messoued* .

January 2025

Abstract

Tunisia's historical legacy is underappreciated due to accessibility challenges and fragmented information. This project introduces 'Tarikhi,' an interactive web service that enables users to explore historic sites via virtual time travel, storytelling, and interactive mapping. Built with FastAPI and SQLite, it integrates secure user authentication and dynamic data visualization. 'Tarikhi' aims to enhance education and appreciation for Tunisia's rich heritage, offering an engaging platform for global audiences

Contents

List of Figures

1 Introduction

Tunisia, with its rich history and cultural diversity, is a treasure trove of historical landmarks that span centuries. The country offers a glimpse into the civilizations that have shaped its identity. However, despite the historical significance of these sites, they often remain underappreciated and inaccessible to a wider audience due to several barriers.

Modern technology provides an opportunity to bridge this gap by creating immersive and interactive ways to experience history. The "Tarikhi" project is an innovative solution that leverages web services to make Tunisia's historical heritage accessible to both local and global audiences. This platform combines storytelling, interactive maps, and time travel features to offer users an educational and engaging journey through Tunisia's past.

"Tarikhi" addresses key challenges such as inconsistent and difficult-to-access information by providing a unified API for historical data. It allows users to virtually explore sites through "Then & Now" visual comparisons, narrated historical tales, and optimized routes for physical or virtual tours. The platform empowers users by enabling them to contribute their stories, share experiences, and enrich the collective knowledge about Tunisia's heritage.

2   Motivation

2.1   Problem Statement

Despite Tunisia's abundance of historic sites, several challenges limit their appreciation and accessibility:

- **Difficult Discovery:** It's challenging for both tourists and researchers to find comprehensive, accurate information about specific sites.

- **Inconsistent Data:** The same historical site may be described differently across various sources, leading to conflicting information and confusion.

- **Limited Access:** Access to information is often restricted by format, language, or location, making it difficult to reach a broad audience.

- **Lack of Historical Visualization and Limited Contribution:** There is no easy way for users to contribute and share their experiences and knowledge with others.

These challenges highlight the need for a solution that combines storytelling, visualization, and user-centered navigation to make Tunisia's history accessible, engaging, and immersive.

2.2   Proposed Solution

To address these challenges, we propose **"Tarikhi: A Traveler's Journey"**, an interactive web service that allows users to explore Tunisia's historical sites through virtual time travel, storytelling, and 3D tours . Key features include:

- **User Empowerment:** A platform to allow registered users to add new content, share their experiences, and collaborate with other users.

- **Unified Data Access:** A single, well-structured API to access detailed information about sites, stories, audio, and artifacts, eliminating the need to search through disparate sources.

- **Before-and-After Visualization:** Users can toggle between historical reconstructions and modern-day images of sites.

- **Secure Access Control:** The API implements JWT authentication and role-based authorization, ensuring that only authorized users can add content or modify sensitive data.

This solution not only enhances the appreciation of Tunisia's history but also provides an educational and entertaining platform for users worldwide. By leveraging modern web technologies such as FastAPI, interactive mapping, and data visualization, our project bridges the gap between Tunisia's rich past and the digital future.

3 Technologies Used

3.1 Backend Development

The project utilizes a modern web framework and programming language for backend development:

- **FastAPI:** A high-performance web framework for building RESTful APIs, enabling rapid development and asynchronous capabilities.

- **Python:** The primary programming language used for the backend logic and API development.

- **API Swagger:** Automatically generated, interactive API documentation provided by FastAPI, making it easy to test and understand available endpoints.

3.2 Database Management

For managing the historical site data and user information, the project employs the following database technologies:

- **SQLite:** A lightweight, serverless database used for storing historical site data, user credentials, and other project-related information.

- **SQLAlchemy:** A powerful Object-Relational Mapping (ORM) tool that simplifies database interactions, making it easier to manage and query data in SQLite.

3.3 Authentication and Security

To ensure secure user authentication and data access, the project implements the following technologies:

- **JSON Web Tokens (JWT):** Used to implement secure authentication and authorization, ensuring that only authenticated users can access or modify protected resources.

3.4 API Testing and Documentation

The project includes tools for testing and documenting the API endpoints:

- **Insomnia:** A collaborative API client used for testing API requests and verifying responses during development.

- **API Swagger:** Interactive API documentation automatically generated by FastAPI, allowing developers to test endpoints directly within the interface.
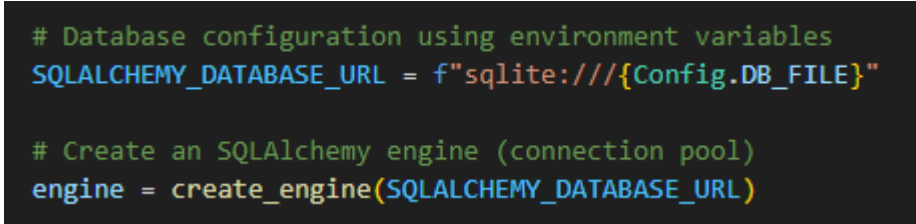
3.5   Deployment

- **Docker:** Docker is used as a containerization platform that packages applications into standardized units for easier development and deployment. It makes it easier to deploy and run the application in different environments, by creating a standardized image to run your application

4 Database

The core of the Tunisian Historical Sites API is a relational database, which acts as a centralized system for storing and organizing all of the different information related to each site, user, stories, audio, and artifacts

4.1 Database Usage

SQLAlchemy ORM: The database interactions are handled through SQLAlchemy, an object-relational mapper that uses Python objects to interact with the data, allowing for complex database queries and data management , which makes the code cleaner and easier to manage. The database engine used by SQLAlchemy is SQLite which is a serverless database system with no configuration ,which is ideal for development and deployment.

```python
# Database configuration using environment variables
SQLALCHEMY_DATABASE_URL = f"sqlite:///{Config.DB_FILE}"

# Create an SQLAlchemy engine (connection pool)
engine = create_engine(SQLALCHEMY_DATABASE_URL)
```

Figure 4.1.1: Database configuration

4.2 database schema

The database schema, using SQLite as the database engine, includes the following tables:

- **sites**: Stores information about historical sites.

- **stories**: Stores narratives associated with sites.

- **audio**: Stores data about audio files related to sites or stories.

- **artifacts**: Stores information about physical objects associated with sites.

- **users**: Stores user authentication and authorization information.

4.3 Database Seeding

The database is initialized with sample data using the 'scripts/db-init.py' script. This script performs the following steps:

1. Creates tables using SQLAlchemy models in a 'sqlite' database.

```python
def create_tables():
    """Creates the tables in the database using SQLAlchemy."""
    try:
        Base.metadata.create_all(engine)
        print("Tables created successfully!")

    except Exception as ex:
        print(f"Error creating tables using sqlalchemy: {ex}")
```

Figure 4.3.1: Creating tables using SQLAlchemy models in SQLite.

2. Populates the tables with initial data for 'sites', 'stories', 'audios', and 'artifacts'.

```python
stories = [
    Story(story_id=uuid.uuid4(), site_id=sites[0].site_id, title="Gladiator Fight",
        summary="A description of a typical gladiator fight in the amphitheater.",
        full_text="A detailed story about the gladiator fight.",
        audio_url="audio_url_gladiator.mp3", author="Historian A", source="Historical Document A",
        type="oral_history"),
    Story(story_id=uuid.uuid4(), site_id=sites[1].site_id, title="A Roman Life",
        summary="A story about how life was in Dougga.",
         full_text="A detailed story of life in Dougga",
        audio_url="audio_url_roman.mp3", author="Historian B", source="Historical Document B",
         type="legend"),
    Story(story_id=uuid.uuid4(), site_id=sites[2].site_id, title="The Legend of Didon",
        summary="The legend of the queen Didon.",
        full_text="A detailed story about the queen Didon.",
        audio_url="audio_url_didon.mp3", author="Historian C", source="Historical Document C",
         type="legend")
]
```

Figure 4.3.2: Seeding the database with sample data.
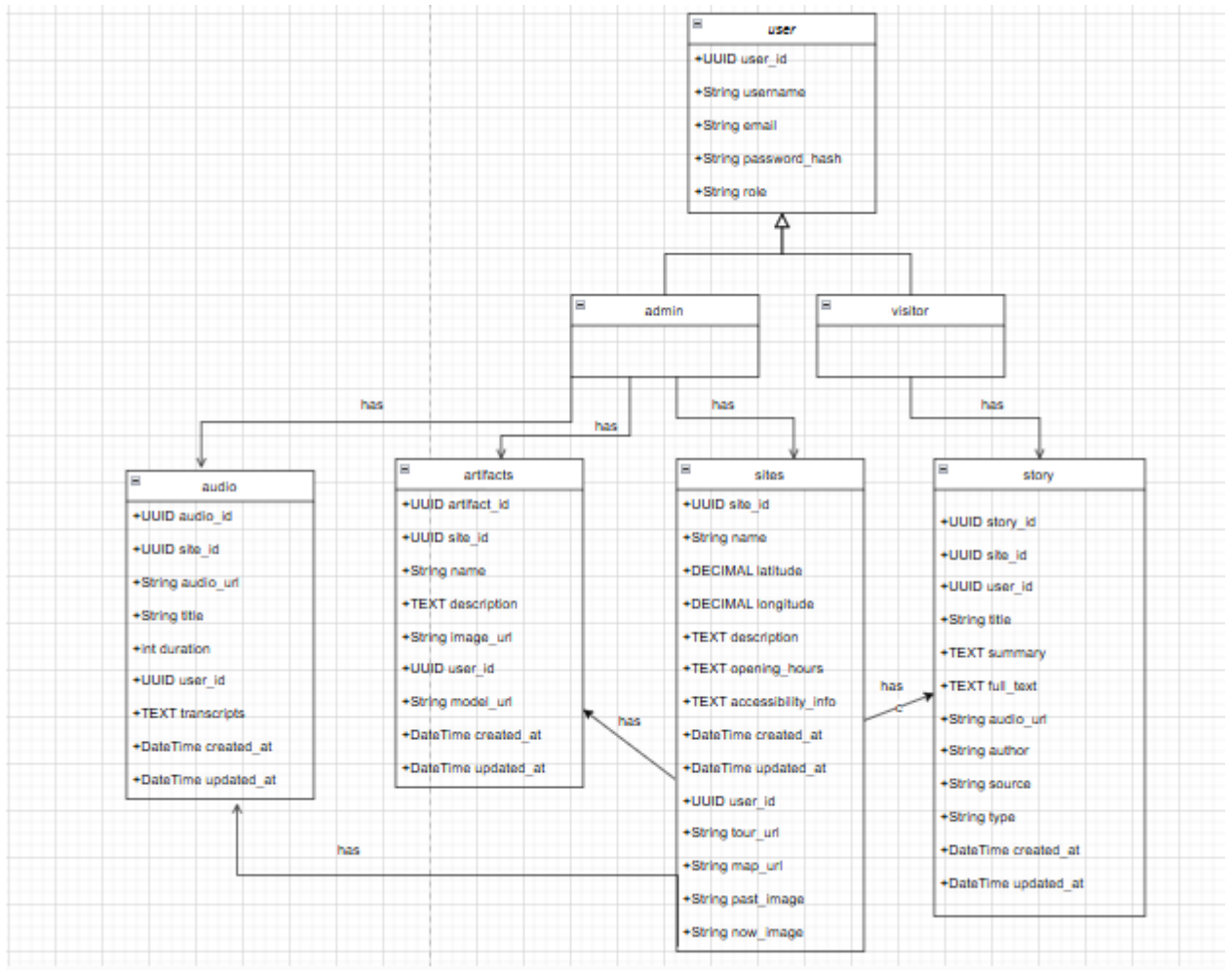
5 High-Level Architecture Diagram

Figure 5.0.1: High-Level Architecture Diagram for the Historic Sites Platform.

6 Security Considerations

6.1 Authentication

JSON Web Token (JWT): The API uses JSON Web Tokens (JWTs) for authentication, implementing a secure way to manage user access to protected resources.

- **JWT Creation**: Access tokens are issued by the '/auth/token' endpoint after a successful authentication using the 'OAuth2Password-RequestForm', and using the 'python-jose' library.

```python
@router.post("/token")
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends(),
                                 db: Session = Depends(database.get_db)):
    user = db.query(User).filter(User.username == form_data.username).first()
    if not user:
        raise HTTPException(status_code = 400, detail = "Incorrect username or password")
    if not user.verify_password(form_data.password):
        raise HTTPException(status_code = 400, detail = "Incorrect username or password")
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username, "role": user.role}, expires_delta=access_token_expires
        )
    return {"access_token": access_token, "token_type": "bearer"}
```

Figure 6.1.1: JWT Creation

- **Token Validation**: The tokens are used to access the protected resources, the server verifies the token using the 'getcurrentuser' function and rejects if the token is not valid.

```python
async def get_current_user(token: str = Depends(OAuth2PasswordBearer(tokenUrl="/auth/token")),
                           db: Session = Depends(database.get_db)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        role : str = payload.get("role")
        if username is None:
            raise HTTPException(status_code = 401, detail= "could not validate credentials")
    except Exception as ex:
        raise HTTPException(status_code = 401, detail= "could not validate credentials")
    user = db.query(User).filter(User.username == username).first()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    user.role = role
    return user
```

Figure 6.1.2: Token Validation

6.2 Authorization

The database is initialized with a pre-configured admin user to facilitate the setup and management of the system. This initial admin entry includes essential details such as a username, a securely hashed password, and a role designation set to "Admin."

```
users = [
 User(username="admin",email="admin@example.com", password_hash=pwd_context.hash("admin*+65@))"),
       role = "admin", user_id=uuid.uuid4())

]
```

Figure 6.2.1: admin information

This initial admin serves as the primary account for accessing restricted endpoints, such as those used for managing site data, creating new entries, or performing administrative tasks. The inclusion of an initial admin ensures that the system is ready for immediate use while maintaining robust access control through role-based permissions.

```
@router.post("/sites", response_model=site_schema.Site, status_code = 201)
async def create_site(site_data: site_schema.SiteCreate, db: Session = Depends(database.get_db),
                          user:User = Depends(get_current_user)):
    try:
      if user.role != "admin":
          raise HTTPException(status_code=403, detail = "User does not have authorization")
      new_site = Site(**site_data.model_dump(), site_id= uuid.uuid4())
      db.add(new_site)
      db.commit()
      db.refresh(new_site)
      return new_site
    except Exception as ex:
      db.rollback()
      raise HTTPException(status_code=500, detail=str(ex))
```

Figure 6.2.2: Authorization

6.3 Password Handling

'passlib' and 'bcrypt': The API uses 'passlib' for password handling, with 'bcrypt' as the hashing algorithm for user passwords before storing them in the database. This ensures that passwords are not stored in plain text.

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class User(Base):
    __tablename__ = "users"

    user_id = Column(UUID(as_uuid = True), primary_key = True, default = uuid.uuid4)
    username = Column(String(255), nullable = False, unique = True)
    email = Column(String(255), nullable = False, unique = True)
    password_hash = Column(String(255), nullable = False)
    created_at = Column(DateTime(timezone=True), server_default = func.now())
    updated_at = Column(DateTime(timezone=True), server_default = func.now(), onupdate = func.now())

    def verify_password(self, password: str) -> bool:
        return pwd_context.verify(password, self.password_hash)

    def hash_password(self, password: str):
        self.password_hash = pwd_context.hash(password)
```

Figure 6.3.1: Password Handling

7 Backend and API Endpoints

Backend Handles all the business logic, including historical site management, story narration, and route optimization. FastAPI is used as the primary framework for developing RESTful APIs.Its integration with Pydantic for data validation helps ensure that only the right data is being used in the applications, and the automated API documentation with Swagger UI provides an interactive way for other developers to access and test the API. This framework enables both fast development and also maintainability

7.1 user Management

- /POST'/users':Creates a new user using the parameters in the body: username, password and email. accessible to general users

- /auth/token: The server verifies the user's credentials and, if valid, responds with an access token. accessible to general users

```
{
    "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiO
iJhZG1pbiIsInJvbGUiOiJhZG1pbiIsImV4cCI6MTczNzE5
MZEwOX0.-6Kjy8BXVyPRbvGJzE5w0JZz9vknhbLVGgEVhfE
pWmM",
    "token_type": "bearer"
}
```

Figure 7.1.1: access token

- /GET'/users':Retrieves all the users stored in the database. Requires Admin role

7.2 Site Management

Endpoints for managing historical sites, including creation, access, and retrieving associated data.

- POST '/sites': Creates a new sites using the parameters in the body. Requires Admin role

- **PUT '/sites/sites-id'**: Updates a site using the information specified in the body. Requires Admin role

- **GET '/sites'**: Retrieves all the sites stored in the database. accessible to general users, requiring at least a basic authenticated role

- **GET '/sites/site-id'**: Retrieves a specific site by its ID, including details about it. accessible to general users, requiring at least a basic authenticated role

- **GET '/sites/name/site-name/description'**: Retrieves the description of a site by its name. accessible to general users, requiring at least a basic authenticated role

```python
@router.get("/sites/name/{site_name}/description", response_model=str)
async def get_site_description_by_name(site_name: str, db: Session = Depends(database.get_db),
                                       user = Depends(get_current_user)):
    try:
        site = db.query(Site).filter(Site.name == site_name).first()
        if site:
            return site.description
        else:
            raise HTTPException(status_code=404, detail="Site not found")
    except Exception as ex:
        raise HTTPException(status_code=500, detail=str(ex))
```

Figure 7.2.1: get description of a site

- **GET '/sites/name/site-name/stories'**: Retrieves all stories related to a specific site using its name.accessible to general users, requiring at least a basic authenticated role

- **GET '/sites/name/site-name/audio'**: Retrieves all audio files related to a site using its name. accessible to general users, requiring at least a basic authenticated role

7.3 Story Management

Endpoints for managing historical narratives and stories associated with sites.

- **GET '/stories'**: Retrieves all the stories stored in the application. accessible to general users, requiring at least a basic authenticated role

- **GET '/stories/story-id'**: Retrieves a specific story by its ID. accessible to general users, requiring at least a basic authenticated role

- **PUT '/stories/story-id'**: Updates a story using the information specified in the body. accessible to general users, requiring at least a basic authenticated role

```python
@router.put("/stories/{story_id}", response_model=story_schema.Story)
async def update_story(story_id: uuid.UUID, story_data: story_schema.StoryCreate,
                       db: Session = Depends(database.get_db),
                       user = Depends(get_current_user)):
    try:
        story = db.query(Story).filter(Story.story_id == story_id).first()
        if story:
            for key, value in story_data.model_dump(exclude_unset = True).items():
                setattr(story,key,value)
            db.commit()
            db.refresh(story)
            return story
        else:
            raise HTTPException(status_code=404, detail="Story not found")

    except Exception as ex:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(ex))
```

Figure 7.3.1: update a story

- **DELETE '/stories/story-id'**: Deletes a story by its ID. accessible to general users, requiring at least a basic authenticated role
- **POST '/stories/name/site-name'**: Creates a new story for a specific site using its name. accessible to general users, requiring at least a basic authenticated role

```python
@router.post("/stories/name/{site_name}", response_model=story_schema.Story, status_code = 201)
async def create_story_by_site_name(site_name: str, story_data: story_schema.StoryCreate,
        db: Session = Depends(database.get_db), user: User = Depends(get_current_user)):
    try:
        site = db.query(Site).filter(Site.name == site_name).first()
        if not site:
            raise HTTPException(status_code=404, detail="Site not found")
        new_story = Story(**story_data.model_dump(exclude={"site_id"}), site_id= site.site_id,
                          user_id = user.user_id, story_id= uuid.uuid4())
        db.add(new_story)
        db.commit()
        db.refresh(new_story)
        return new_story
    except Exception as ex:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(ex))
```

Figure 7.3.2: create a story

7.4 Audio Management

Endpoints for managing audio files associated with historical sites.

- **GET '/audio'**: Retrieves all audio files stored in the application. accessible to general users, requiring at least a basic authenticated role

```
@router.get("/audio", response_model=List[audio_schema.Audio])
async def get_all_audio_files(db: Session = Depends(database.get_db),
                              user = Depends(get_current_user)):
    try:
        audio_files = db.query(Audio).all()
        return audio_files
    except Exception as ex:
        raise HTTPException(status_code=500, detail=str(ex))
```

Figure 7.4.1: get audios

- **PUT '/audio/audio-id'**: Updates a specific audio file by its ID. Requires Admin role

- **DELETE '/audio/audio-id'**: Deletes an audio file by its ID. Requires Admin role

- **POST '/audio/name/site-name'**: Creates a new audio file for a specific site using its name. Requires Admin role

7.5 Artifact Management

Endpoints for managing artifacts found at or related to historical sites.

- **GET '/artifacts'**: Retrieves all artifacts stored in the application. accessible to general users, requiring at least a basic authenticated role

- **POST '/artifacts'**: Creates a new artifact object using the parameters in the body. Requires Admin role

- **PUT '/artifacts/artifact-id'**: Updates a specific artifact using the specified ID and new values. Requires Admin role

- **DELETE '/artifacts/artifact-id'**: Deletes a specific artifact object by its ID. Requires Admin role

- **POST '/artifacts/name/site-name'**: Creates a new artifact for a specific site using its name and data in the body. Requires Admin role

7.6 Search

Allows users to search specific information throughout the database using keywords.

- **GET '/search'**: Allows users to search across the database for sites, stories, and audio using a free-text query. accessible to general users, requiring at least a basic authenticated role

```python
@router.get("/search", response_model=List[dict])
async def search_all(q: str, db: Session = Depends(database.get_db), user=Depends(get_current_user)):
    try:

        search_params = (f"%{q}%",) * 7


        results = db.execute(text("""
            SELECT 'site' AS type, site_id, name as title, description as summary
            FROM sites
            WHERE LOWER(name) LIKE LOWER(:search) OR LOWER(description) LIKE LOWER(:search)
            UNION ALL
            SELECT 'story' AS type, story_id, title, summary
            FROM stories
            WHERE LOWER(title) LIKE LOWER(:search) OR LOWER(summary) LIKE LOWER(:search) OR
                                LOWER(full_text) LIKE LOWER(:search)
            UNION ALL
            SELECT 'audio' AS type, audio_id, title, transcripts
            FROM audio
            WHERE LOWER(title) LIKE LOWER(:search) OR LOWER(transcripts) LIKE LOWER(:search)
        """), {"search": f"%{q}%"}).fetchall()


        search_list = [
            {"type": result[0], "id": result[1], "title": result[2], "summary": result[3]}
            for result in results
        ]

        return search_list

    except Exception as ex:
        raise HTTPException(status_code=500, detail=str(ex))
```

Figure 7.6.1: search

8 Docker

Docker containerize the FastAPI application, ensuring a consistent and reproducible environment across different stages of development, testing, and deployment. Docker encapsulates the application along with all its dependencies, configurations, and run-time environment, and also allows you to generate a shareable image.
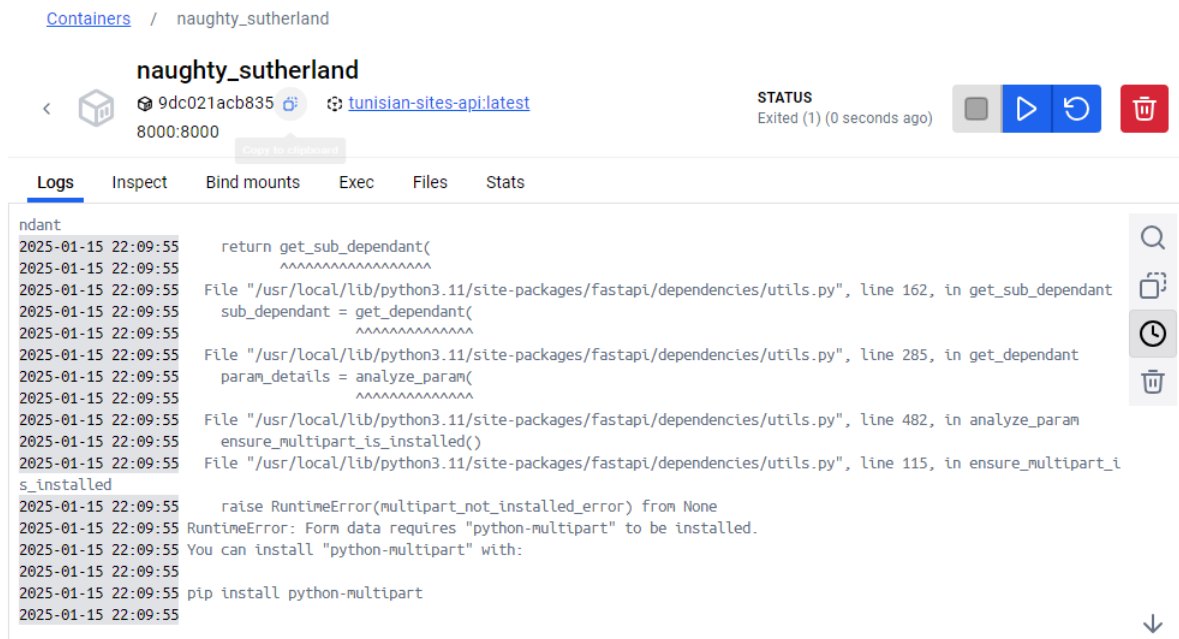

Figure 8.0.1: docker container

Figure 8.0.2: docker image

9 Conclusion

The "Tarikhi" project transforms how Tunisia's rich historical heritage is accessed and appreciated, addressing challenges like fragmented information and limited engagement. By combining virtual time travel, storytelling, and interactive mapping, it offers an immersive, educational experience for users worldwide. Built with modern technologies like FastAPI and SQLite, the platform ensures reliability and scalability while empowering users to contribute content. Looking forward, "Tarikhi" sets the stage for broader applications and deeper integration of advanced technologies, redefining how we connect with history in the digital age.

Prospects

The "Tarikhi" platform opens up numerous opportunities for enhancing the accessibility and appreciation of Tunisia's historical heritage. Some of the key prospects include:

- **Multilingual Support:**

  - Translate the platform into multiple languages (e.g., French, English, Arabic) to reach a broader global audience.
  - Enable dynamic switching between languages for a seamless user experience.

- **3D Virtual Tours:**

  - Integrate 3D models of historical sites to provide immersive exploration.
  - Use augmented reality (AR) for "on-site" experiences through mobile devices.

- **Integration with AI:**

  - Implement AI for automatic historical image restoration or reconstruction.
  - Use natural language processing (NLP) for generating dynamic, engaging storytelling.

- **Educational Institutions Partnerships:**

  - Partner with schools and universities to provide exclusive educational modules.
  - Offer the platform as a resource for history or tourism curricula.

The prospects for "Tarikhi" are vast, positioning it as a transformative tool for preserving and promoting historical heritage in the digital age.

References

1. Jeffrey K. Pinto, *Project Management: Achieving Competitive Advantage* (5th Edition), 2020.

2. FastAPI Official Documentation. Available at: `https://fastapi.tiangolo.com/`

3. SQLAlchemy Official Documentation. Available at: `https://www.sqlalchemy.org/`

4. Python Official Documentation. Available at: `https://www.python.org/`

5. SQLite Official Documentation. Available at: `https://www.sqlite.org/`

6. Insomnia Official Website. Available at: `https://www.insomnia.com/`

7. Swagger UI Documentation. Available at: `https://swagger.io/tools/swagger-ui/`

8. Docker Official Documentation. Available at: `https://docs.docker.com/`