### ThinkEdge - Complete Backend Source Code

This document contains the full source code for the Python backend server, organized by file path.

### 1. requirements.txt

This file lists all the necessary Python libraries for the project.

fastapi  
uvicorn[standard]  
pydantic  
python-dotenv  
google-cloud-firestore  
google-generativeai  
passlib[bcrypt]  
python-jose[cryptography]

### 2. main.py

This is the main entry point that creates the FastAPI application and includes the API routers.

# /app/main.py  
  
from fastapi import FastAPI  
from app.api.v1.endpoints import users, lessons, gemini  
from app.core.config import settings  
  
# Initialize the FastAPI app  
app = FastAPI(  
 title=settings.PROJECT\_NAME,  
 openapi\_url=f"{settings.API\_V1\_STR}/openapi.json"  
)  
  
# Include the API routers for different functionalities  
app.include\_router(users.router, prefix=settings.API\_V1\_STR, tags=["users"])  
app.include\_router(lessons.router, prefix=settings.API\_V1\_STR, tags=["lessons"])  
app.include\_router(gemini.router, prefix=settings.API\_V1\_STR, tags=["gemini"])  
  
@app.get("/")  
def read\_root():  
 """  
 Root endpoint for health checks.  
 """  
 return {"message": "Welcome to the ThinkEdge API"}

### 3. core/config.py

This file manages all configuration settings for the application, loading them from environment variables.

# /app/core/config.py  
  
import os  
from dotenv import load\_dotenv  
  
# Load environment variables from a .env file  
load\_dotenv()  
  
class Settings:  
 PROJECT\_NAME: str = "ThinkEdge AI Learning Platform"  
 API\_V1\_STR: str = "/api/v1"  
  
 # Gemini API Configuration  
 GEMINI\_API\_KEY: str = os.getenv("GEMINI\_API\_KEY")  
  
 # Firebase/Firestore Configuration  
 # The path to your Firebase service account JSON file  
 FIREBASE\_SERVICE\_ACCOUNT\_KEY\_PATH: str = os.getenv("FIREBASE\_SERVICE\_ACCOUNT\_KEY\_PATH")  
  
 # Security settings for JWT  
 SECRET\_KEY: str = os.getenv("SECRET\_KEY", "a\_very\_secret\_key\_for\_development")  
 ALGORITHM: str = "HS256"  
 ACCESS\_TOKEN\_EXPIRE\_MINUTES: int = 60 \* 24 \* 8 # 8 days  
  
settings = Settings()

### 4. api/v1/schemas.py

This file contains all the Pydantic models used for data validation and serialization in the API.

# /app/api/v1/schemas.py  
  
from pydantic import BaseModel, EmailStr  
from typing import Optional, List  
  
# --- User Schemas ---  
class UserBase(BaseModel):  
 email: EmailStr  
 display\_name: Optional[str] = None  
  
class UserCreate(UserBase):  
 password: str  
  
class User(UserBase):  
 id: str  
  
 class Config:  
 orm\_mode = True  
  
# --- Token Schemas ---  
class Token(BaseModel):  
 access\_token: str  
 token\_type: str  
  
class TokenData(BaseModel):  
 email: Optional[EmailStr] = None  
  
# --- Chat Schemas ---  
class ChatMessage(BaseModel):  
 text: str  
 sender: str # "user" or "ai"

### 5. core/security.py

This file handles all security-related logic, including password hashing and JWT token creation/verification.

# /app/core/security.py  
  
from datetime import datetime, timedelta  
from typing import Optional  
from jose import JWTError, jwt  
from passlib.context import CryptContext  
from app.core.config import settings  
  
# Setup password hashing  
pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")  
  
def verify\_password(plain\_password: str, hashed\_password: str) -> bool:  
 """Verifies a plain password against a hashed one."""  
 return pwd\_context.verify(plain\_password, hashed\_password)  
  
def get\_password\_hash(password: str) -> str:  
 """Hashes a plain password."""  
 return pwd\_context.hash(password)  
  
def create\_access\_token(data: dict, expires\_delta: Optional[timedelta] = None):  
 """Creates a new JWT access token."""  
 to\_encode = data.copy()  
 if expires\_delta:  
 expire = datetime.utcnow() + expires\_delta  
 else:  
 expire = datetime.utcnow() + timedelta(minutes=settings.ACCESS\_TOKEN\_EXPIRE\_MINUTES)  
 to\_encode.update({"exp": expire})  
 encoded\_jwt = jwt.encode(to\_encode, settings.SECRET\_KEY, algorithm=settings.ALGORITHM)  
 return encoded\_jwt  
  
# Placeholder for getting the current user from a token  
# In a real app, this would verify the token from the request header  
async def get\_current\_user(token: str = "placeholder\_token"):  
 # This is a placeholder. A real implementation would decode and validate the token.  
 # For now, it returns a dummy user for demonstration purposes.  
 from app.api.schemas import User  
 return User(id="dummy\_user\_id", email="test@example.com", display\_name="Test User")

### 6. services/firestore\_service.py

This service contains all the logic for interacting with the Firestore database.

# /app/services/firestore\_service.py  
  
import google.cloud.firestore  
from app.core.config import settings  
  
# Initialize Firestore client  
# This assumes you have set up Google Cloud authentication in your environment  
# (e.g., by setting the GOOGLE\_APPLICATION\_CREDENTIALS environment variable)  
db = google.cloud.firestore.Client.from\_service\_account\_json(  
 settings.FIREBASE\_SERVICE\_ACCOUNT\_KEY\_PATH  
)  
  
def get\_user\_chat\_history(user\_id: str) -> List[dict]:  
 """  
 Retrieves the chat history for a specific user from Firestore.  
 In a real app, this would be more complex, handling pagination etc.  
 """  
 # Placeholder implementation  
 # A real implementation would fetch messages from a 'chats' collection  
 # and order them by timestamp.  
 print(f"Fetching chat history for user: {user\_id}")  
 return [  
 {"role": "user", "parts": [{"text": "Hello, Sensei!"}]},  
 {"role": "model", "parts": [{"text": "Hello! How can I help you with your Japanese studies today?"}]}  
 ]  
  
def add\_message\_to\_history(user\_id: str, message: dict):  
 """  
 Adds a new message to a user's chat history in Firestore.  
 """  
 # Placeholder implementation  
 # A real implementation would add a new document to a subcollection  
 # under the user's chat history.  
 print(f"Adding message for user {user\_id}: {message}")  
 pass

### 7. services/gemini\_service.py

This service contains the core logic for interacting with the Gemini API.

# /app/services/gemini\_service.py  
  
import google.generativeai as genai  
from app.core.config import settings  
from app.services.firestore\_service import get\_user\_chat\_history, add\_message\_to\_history  
  
# Configure the Gemini API client  
genai.configure(api\_key=settings.GEMINI\_API\_KEY)  
  
# Select the model  
model = genai.GenerativeModel('gemini-1.5-flash')  
  
async def get\_conversational\_response(user\_id: str, user\_message: str) -> str:  
 """  
 Gets a conversational response from Gemini, using the user's chat history.  
 """  
 try:  
 # 1. Get the user's existing chat history from Firestore  
 chat\_history = get\_user\_chat\_history(user\_id)  
  
 # 2. Start a chat session with the existing history  
 chat = model.start\_chat(history=chat\_history)  
  
 # 3. Send the new message to Gemini  
 response = await chat.send\_message\_async(user\_message)  
 ai\_response\_text = response.text  
  
 # 4. Save the new user message and AI response back to Firestore  
 add\_message\_to\_history(user\_id, {"role": "user", "parts": [{"text": user\_message}]})  
 add\_message\_to\_history(user\_id, {"role": "model", "parts": [{"text": ai\_response\_text}]})  
  
 return ai\_response\_text  
  
 except Exception as e:  
 print(f"An error occurred while interacting with Gemini API: {e}")  
 return "I'm sorry, I'm having trouble connecting right now. Please try again later."

### 8. api/v1/endpoints/gemini.py

This file defines the actual API endpoint that the mobile app will call to interact with the AI.

# /app/api/v1/endpoints/gemini.py  
  
from fastapi import APIRouter, Depends, HTTPException  
from app.core.security import get\_current\_user  
from app.services.gemini\_service import get\_conversational\_response  
from app.api.schemas import User, ChatMessage  
  
router = APIRouter()  
  
@router.post("/chat", response\_model=ChatMessage)  
async def chat\_with\_ai\_tutor(  
 message: ChatMessage,  
 current\_user: User = Depends(get\_current\_user)  
):  
 """  
 Endpoint for conversational practice with the AI Tutor.  
 """  
 if not message.text:  
 raise HTTPException(status\_code=400, detail="Message text cannot be empty.")  
  
 try:  
 ai\_response\_text = await get\_conversational\_response(  
 user\_id=current\_user.id,  
 user\_message=message.text  
 )  
 return ChatMessage(text=ai\_response\_text, sender="ai")  
 except Exception as e:  
 # In a production app, we would log this error properly  
 print(f"Error in /chat endpoint: {e}")  
 raise HTTPException(status\_code=500, detail="An error occurred while processing your request.")

### 9. Placeholder Endpoints (users.py, lessons.py)

These files are included to show the complete structure but contain placeholder logic.

# /app/api/v1/endpoints/users.py  
  
from fastapi import APIRouter  
from app.api.schemas import User  
  
router = APIRouter()  
  
@router.get("/users/me", response\_model=User)  
async def read\_users\_me():  
 """  
 Placeholder endpoint to get the current user's profile.  
 """  
 return {"id": "dummy\_user\_id", "email": "test@example.com", "display\_name": "Test User"}

# /app/api/v1/endpoints/lessons.py  
  
from fastapi import APIRouter  
from typing import List  
  
router = APIRouter()  
  
@router.get("/lessons/{level}")  
async def get\_lessons\_for\_level(level: str) -> List:  
 """  
 Placeholder endpoint to get lessons for a specific JLPT level.  
 """  
 return [{"id": "lesson1", "title": f"First Lesson of {level}"}]