Lab 3 Report: Building a Microservices Architecture with API Gateway using Python FastAPI

IT4020: Modern Topics in IT

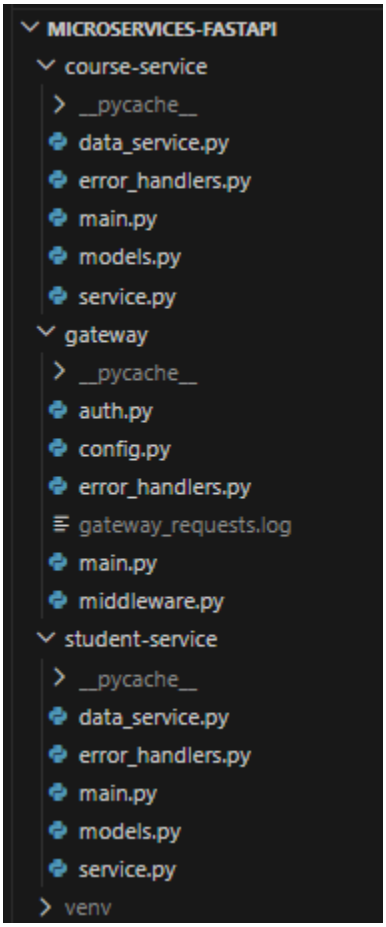
Submitted By: Rathnayaka. H. M. S. S IT22277190

Date of Submission: 24.02.2026

Student Summary of Completed Tasks

- Set up a FastAPI-based microservices project with isolated services.
- Implemented Student Microservice with full CRUD.
- Implemented API Gateway with request forwarding.
- Added Course Microservice and integrated it with the gateway.
- Added JWT-based authentication for protected gateway routes.
- Added request logging middleware in the gateway.
- Added structured, service-specific error handling.
- Ran direct-service and gateway-level endpoint tests.

File Structure



Git Repository : https://github.com/IT22277190/MTIT_Practical-3_microservices-.git

Step 1: Project Setup and Dependencies

Step Objective

Initialize the project environment and install all required dependencies for FastAPI microservices, gateway routing, authentication, and logging.

Reference Code

```
# requirements.txt
```

```
fastapi==0.115.0
uvicorn[standard]==0.32.0
pydantic==2.10.0
httpx==0.27.0
python-multipart==0.0.12
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4
python-dotenv==1.0.0
```

Terminal Inputs

```
python -m venv venv
venv\Scripts\activate
pip install -r requirements.txt
```

Terminal Outputs

```
Successfully installed fastapi uvicorn pydantic httpx python-multipart python-jose passlib
bcrypt python-dotenv ...
```

```
Successfully installed packages from requirements.txt
```

Explanation

This establishes a reproducible environment and includes all libraries required by the base lab and all extension activities.

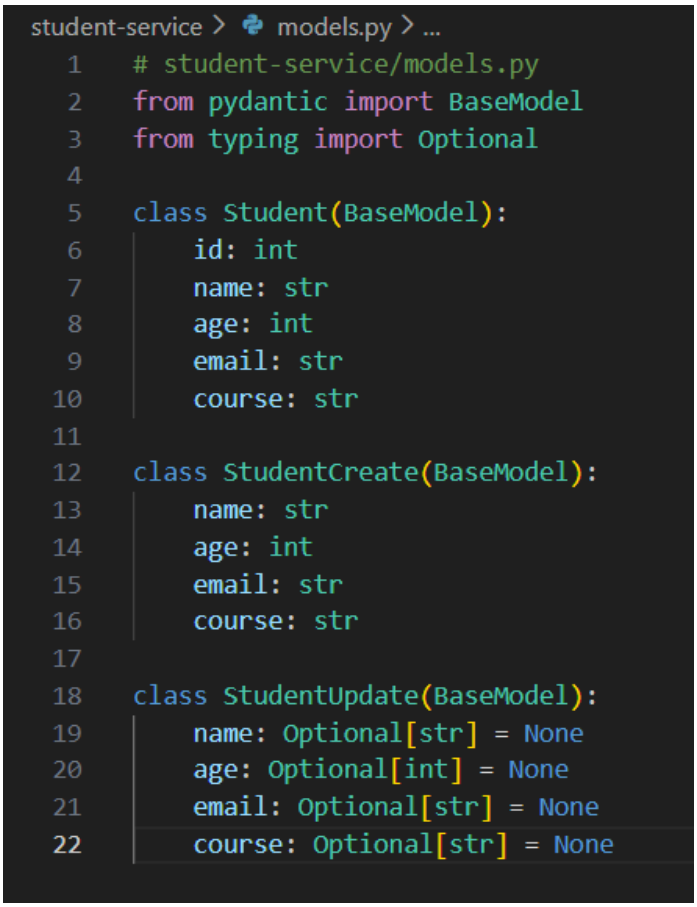
Step 2: Student Microservice Data Models

Step Objective

Implement student schemas and full CRUD API endpoints by Defining request/response schemas for student operations with strong validation via Pydantic.

Reference Code

```
# student-service/models.py
```



Explanation

Separate models for create/update avoid accidental ID tampering and support partial updates cleanly.

Step 3: Student Mock Data Layer

Step Objective

Create an in-memory repository to simulate persistence and enable complete CRUD without a database.

Reference Code

```
# student-service/data_service.py
```

student-service > data_service.py > ...

```python
# student-service/data_service.py
from models import Student

class StudentMockDataService:
    def __init__(self):
        self.students = [
            Student(id=1, name="John Doe", age=20, email="john@example.com", course="Computer Science"),
            Student(id=2, name="Jane Smith", age=22, email="jane@example.com", course="Information Technology"),
            Student(id=3, name="Bob Johnson", age=21, email="bob@example.com", course="Software Engineering"),
        ]
        self.next_id = 4

    def get_all_students(self):
        return self.students

    def get_student_by_id(self, student_id: int):
        return next((s for s in self.students if s.id == student_id), None)

    def add_student(self, student_data):
        new_student = Student(id=self.next_id, **student_data.dict())
        self.students.append(new_student)
        self.next_id += 1
        return new_student

    def update_student(self, student_id: int, student_data):
        student = self.get_student_by_id(student_id)
        if student:
            update_data = student_data.dict(exclude_unset=True)
            for key, value in update_data.items():
                setattr(student, key, value)
            return student
        return None

    def delete_student(self, student_id: int):
        student = self.get_student_by_id(student_id)
        if student:
            self.students.remove(student)
            return True
        return False
```

Explanation

This layer centralizes data access logic and keeps API/controller code clean.

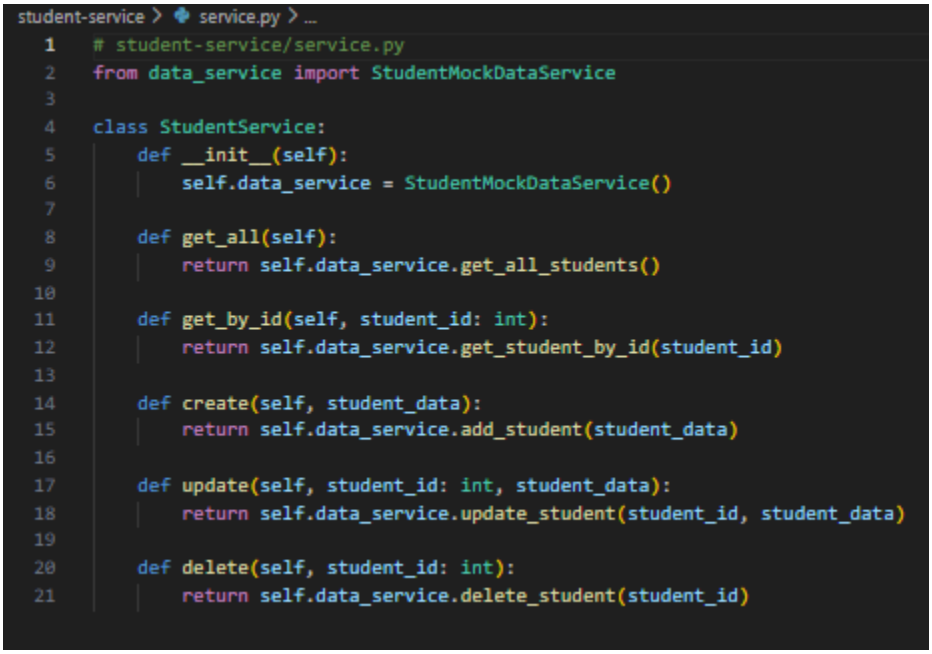
Step 4: Student Service Business Layer

Step Objective

Introduce a service layer to separate business operations from API routes.

Reference Code

```
# student-service/service.py
```



Explanation

A dedicated service layer improves maintainability and follows clean architecture principles.

Step 5: Student FastAPI Application (CRUD Endpoints)

Step Objective

Expose REST endpoints for student CRUD operations using FastAPI.

Reference Code

```
# student-service/main.py

from fastapi import FastAPI, HTTPException, status

from fastapi.exceptions import RequestValidationError

from starlette.exceptions import HTTPException as StarletteHTTPException

from models import Student, StudentCreate, StudentUpdate

from service import StudentService

from typing import List

from error_handlers import (

    http_exception_handler, validation_exception_handler, general_exception_handler
```

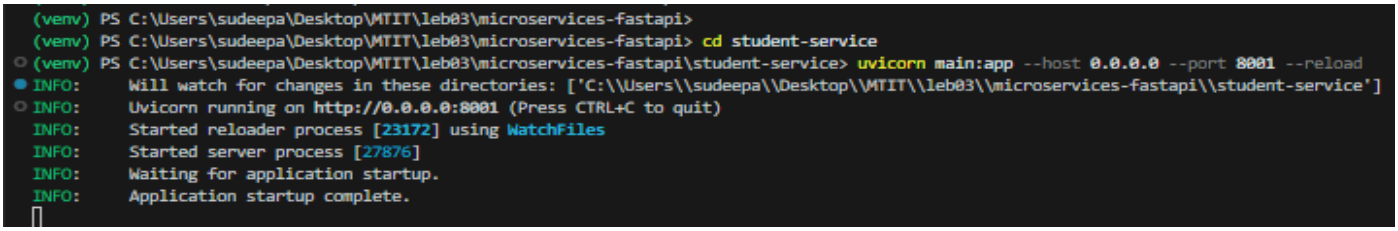
```
)

app = FastAPI(title="Student Microservice", version="1.0.0")

# Register exception handlers
app.add_exception_handler(StarletteHTTPException, http_exception_handler)
app.add_exception_handler(RequestValidationError, validation_exception_handler)
app.add_exception_handler(Exception, general_exception_handler)

student_service = StudentService()

@app.get("/")
def read_root():
    return {"message": "Student Microservice is running"}

@app.get("/api/students", response_model=List[Student])
def get_all_students():
    return student_service.get_all()

@app.get("/api/students/{student_id}", response_model=Student)
def get_student(student_id: int):
    student = student_service.get_by_id(student_id)
    if not student:
        raise HTTPException(status_code=404, detail="Student not found")
    return student

@app.post("/api/students", response_model=Student, status_code=status.HTTP_201_CREATED)
def create_student(student: StudentCreate):
    return student_service.create(student)

@app.put("/api/students/{student_id}", response_model=Student)
```

```
def update_student(student_id: int, student: StudentUpdate):
    updated_student = student_service.update(student_id, student)
    if not updated_student:
        raise HTTPException(status_code=404, detail="Student not found")
    return updated_student


@app.delete("/api/students/{student_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_student(student_id: int):
    success = student_service.delete(student_id)
    if not success:
        raise HTTPException(status_code=404, detail="Student not found")
    return None
```

Terminal Inputs

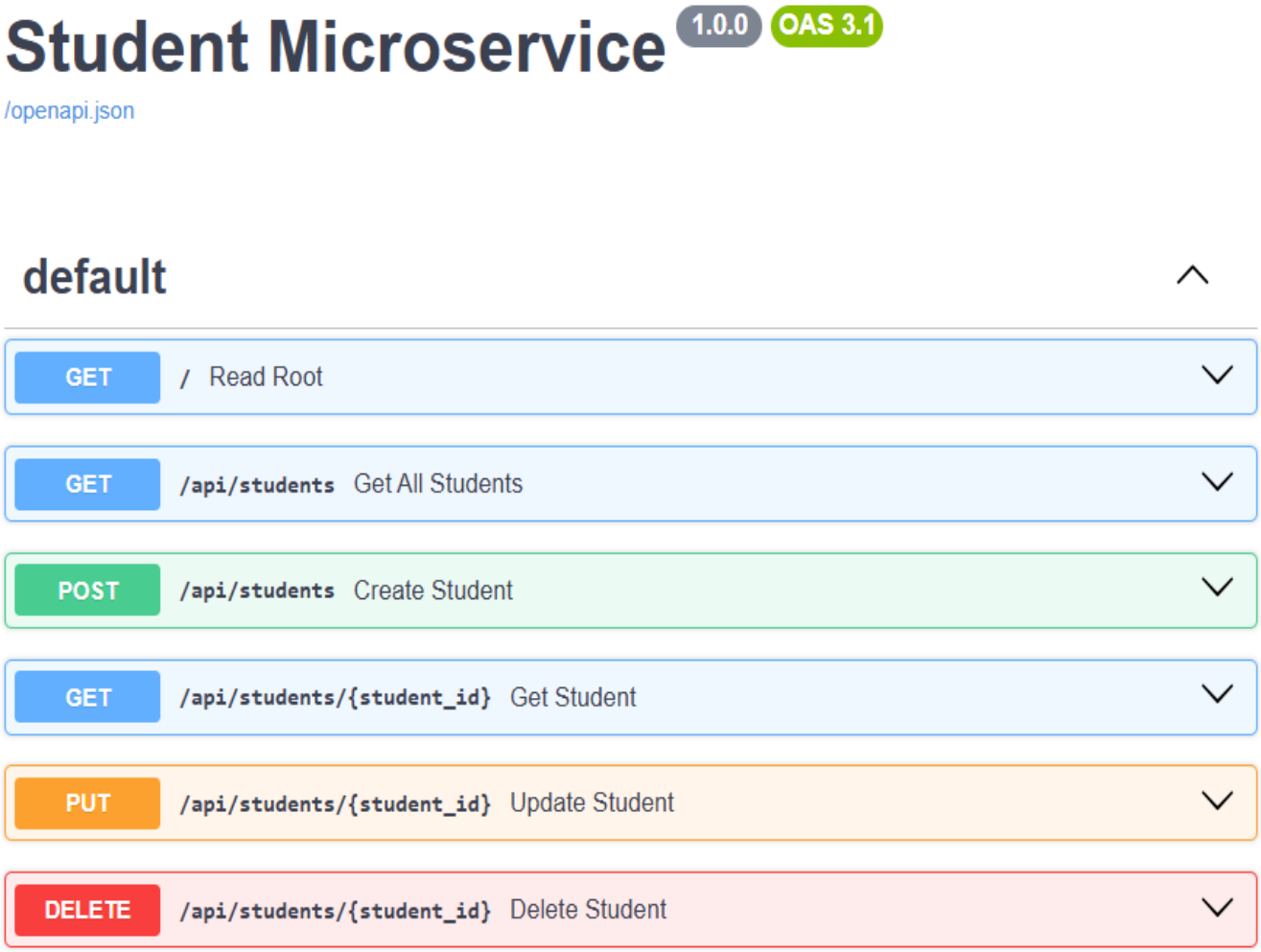
```
cd student-service

../venv/Scripts/python.exe -m uvicorn main:app --port 8001
```

Terminal Outputs



Explanation

This implements the core microservice required by the lab and provides all CRUD operations.



Step 6: API Gateway Base Routing

Step Objective

Implement an API gateway that forwards incoming requests to target microservices.

```
# gateway/main.py (core)
SERVICES = {
    "student": "http://localhost:8001",
    "course": "http://localhost:8002"
}

async def forward_request(service: str, path: str, method: str, **kwargs):
```

Reference Code

```
# (Core forwarding logic from gateway/main.py)
```

```python
# gateway/main.py (core)
SERVICES = {
    "student": "http://localhost:8001",
    "course": "http://localhost:8002"
}

async def forward_request(service: str, path: str, method: str, **kwargs):
```

Terminal Inputs

```
../venv/Scripts/python.exe -m uvicorn main:app --port 8000
```

```
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi> cd gateway
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi\gateway>
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi\gateway>
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi\gateway>
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi\gateway> uvicorn main:app --host 0.0.0.0 --port 8000 --reload
INFO:     Will watch for changes in these directories: ['C:\\Users\\sudeepa\\Desktop\\MTIT\\leb03\\microservices-fastapi\\gateway']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [26708] using WatchFiles
INFO:     Started server process [27348]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Terminal Outputs

```
INFO:     Uvicorn running on http://127.0.0.1:8000
```

Explanation

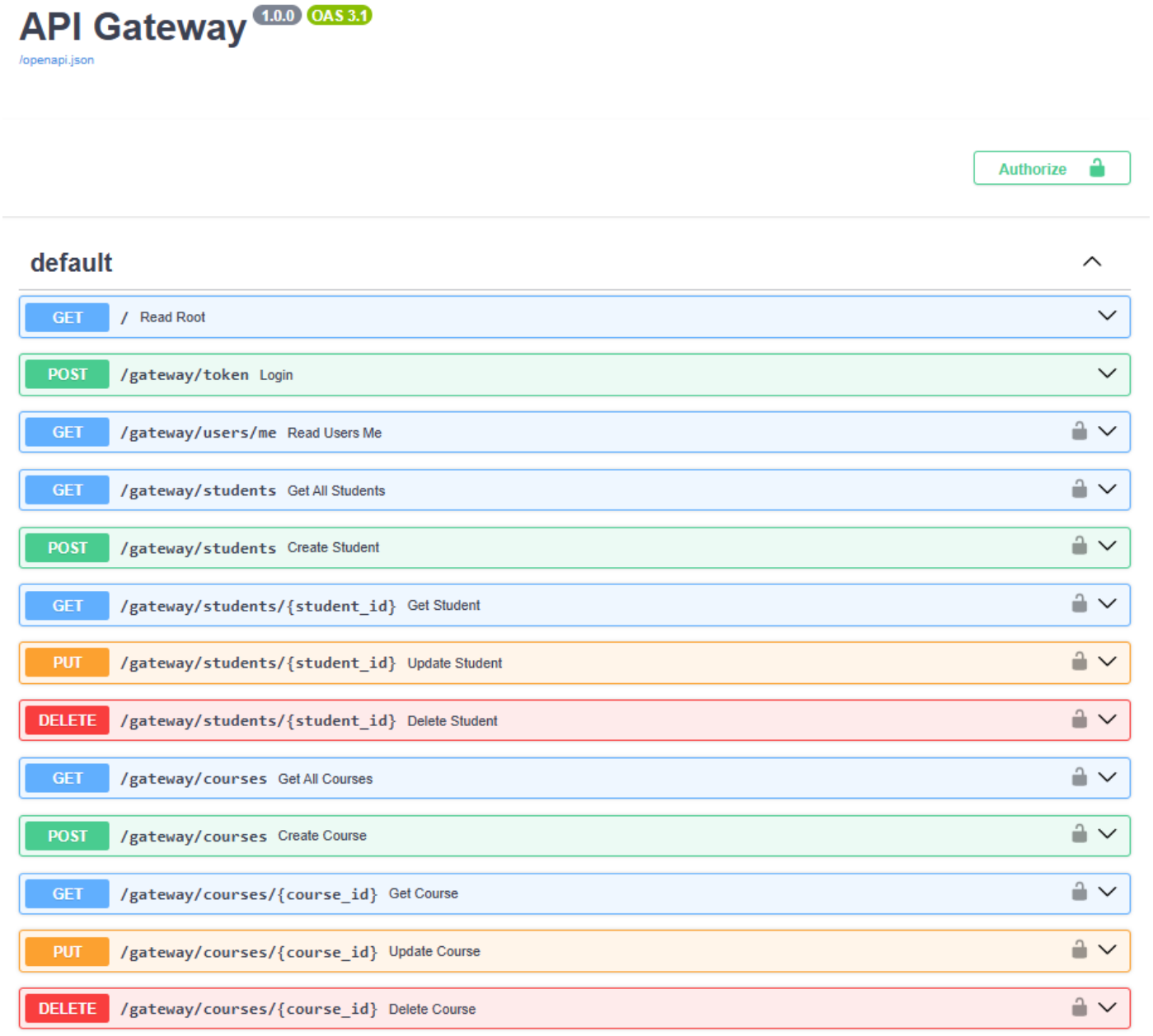
The gateway becomes the single entry point for clients, an essential API Gateway pattern objective.

Gateway Update Note (After Creating Course Service)

After the Course Service was created, the API Gateway also had to be updated so it could route course-related requests. This required adding a new `course` entry to the gateway `SERVICES` map and defining `/gateway/courses` endpoints for GET, POST, PUT, and DELETE operations. This ensures that both microservices—Student and Course—become accessible through the unified gateway entry point.

Before vs After (Gateway Integration)

Area	Before Course Service	After Course Service
`SERVICES` map in gateway	`student` only	`student` + `course`
Gateway route groups	`/gateway/students/*` only	`/gateway/students/*` and `/gateway/courses/*`
Running service processes	Student + Gateway	Student + Course + Gateway
End-to-end test scope	Student flow only	Student + Course flows through gateway



Step 7: Activity 1 - Course Microservice + Gateway Integration

Step Objective

Add a new independent Course microservice on port 8002 and expose it via gateway routes.

Reference Codes

```
# course-service/models.py
```

course-service > models.py > ...

```python
# course-service/models.py
from pydantic import BaseModel
from typing import Optional

class Course(BaseModel):
    id: int
    code: str
    name: str
    credits: int
    instructor: str
    department: str

class CourseCreate(BaseModel):
    code: str
    name: str
    credits: int
    instructor: str
    department: str

class CourseUpdate(BaseModel):
    code: Optional[str] = None
    name: Optional[str] = None
    credits: Optional[int] = None
    instructor: Optional[str] = None
    department: Optional[str] = None

```

Explanation

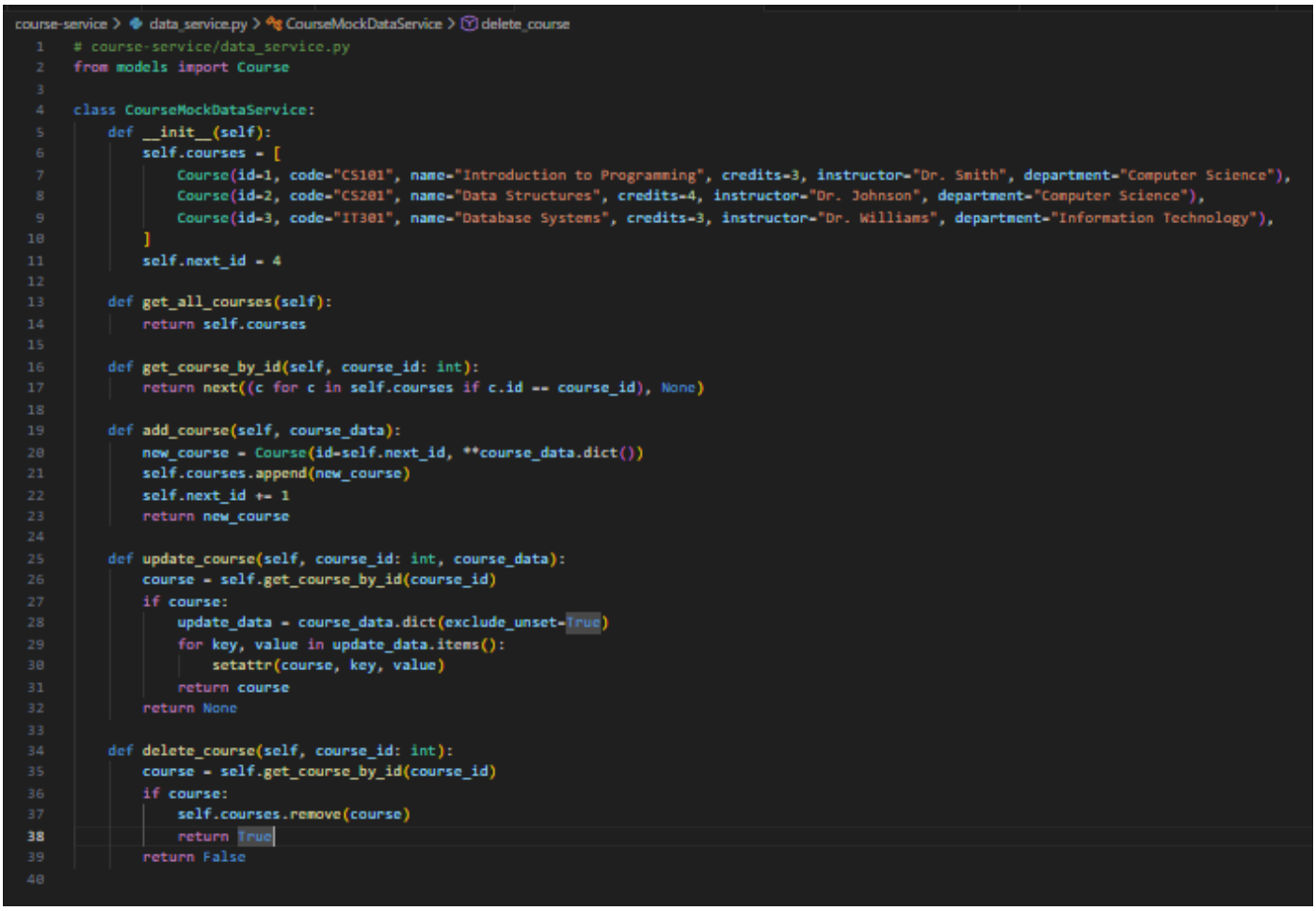
`Service.py` is a **wrapper layer** that delegates all CRUD operations to `CourseMockDataService`, making your code more modular and easier to extend.

```
# course-service/main.py
```

Explanation

This file serves as the **first validation boundary**, ensuring that incoming request payloads conform to expected structure and data types *before* they reach the business logic layer.

```
# course-service/data_service.py
```



Explanation

This class is a **mock database service** for courses:

- **Read:** get_all_courses, get_course_by_id
- **Create:** add_course
- **Update:** update_course
- **Delete:** delete_course

It's useful for testing APIs or applications without needing a real database.

Terminal Inputs

```
cd course-service
```

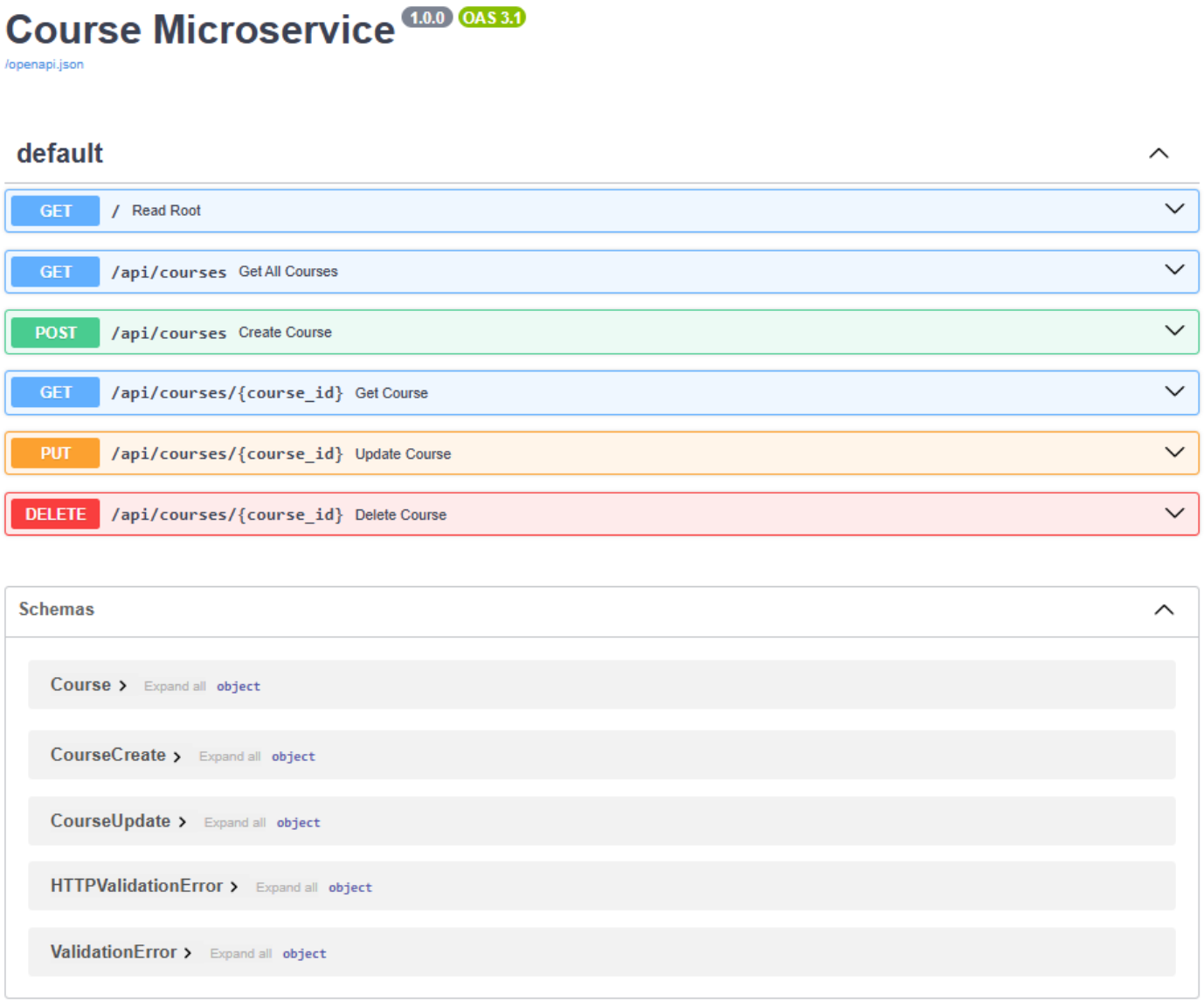
```
../venv/Scripts/python.exe -m uvicorn main:app --port 8002
```

Terminal Outputs

```
(venv) PS C:\Users\sudeepa\Desktop\MTIT\leb03\microservices-fastapi\course-service> uvicorn main:app --host 0.0.0.0 --port 8002 --reload
INFO:     Will watch for changes in these directories: ['C:\\Users\\sudeepa\\Desktop\\MTIT\\leb03\\microservices-fastapi\\course-service']
INFO:     Uvicorn running on http://0.0.0.0:8002 (Press CTRL+C to quit)
INFO:     Started reloader process [10532] using WatchFiles
INFO:     Started server process [14848]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Explanation

This demonstrates extensibility of the architecture by adding another independent domain service.



Step 8: Activity 2 - JWT Authentication in Gateway

Step Objective

Secure gateway endpoints using JWT so only authenticated users can access microservice routes.

Reference Code

`# gateway/auth.py (key sections)`

Tools and settings needed for authentication:

```
from jose import JWTError, jwt
from passlib.context import CryptContext
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from pydantic import BaseModel
from config import SECRET_KEY, ALGORITHM, ACCESS_TOKEN_EXPIRE_MINUTES, MOCK_USERS
```

- **JWTs** for secure token-based login.
- **Password hashing** for credential safety.
- **FastAPI utilities** for error handling and dependencies.
- **OAuth2** for login/token flow.
- **Pydantic models** for structured data.
- **Config values** for security and mock user storage

Password Hashing & OAuth2 Scheme

```
# Password hashing
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# OAuth2 scheme
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="gateway/token")
```

- Uses bcrypt to hash and verify passwords.
- Ensures secure storage of user credentials.
- Defines the authentication scheme.
- `tokenUrl="gateway/token"` means clients must request tokens from `/gateway/token`.

Dependency Functions

These are used in FastAPI routes to enforce authentication.

```python
async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception
    user = get_user(username=token_data.username)
    if user is None:
        raise credentials_exception
    return user
```

- get_current_user:
- Decodes JWT token.
- Extracts `username`.
- Retrieves user from `MOCK_USERS`.
- Raises `HTTPException` if invalid.

get_current_active_user:

```python
async def get_current_active_user(current_user: User = Depends(get_current_user)):
    if current_user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user

```

- Ensures the user is not disabled.
- Raises error if inactive.

login attempt and runtime behavior

Step 1: Login Attempt

```
Invoke-RestMethod -Uri "http://127.0.0.1:8000/gateway/token" -Method Post -ContentType "application/x-www-form-urlencoded" -Body "username=admin&password=secret"
```

- This sends a POST request to `/gateway/token`.
- The body contains `username=admin` and `password=secret`.
- This is the standard OAuth2 password flow in FastAPI

Step 2: Error Response

```
Invoke-RestMethod : {"error":"Internal Server Error","detail":"An unexpected error occurred","status_code":500,"path":"http://127.0.0.1:8000/gateway/token","method":"POST","error_type":"ValueError"}
```

- The server returned 500 Internal Server Error.
- Root cause: from the bcrypt backend.

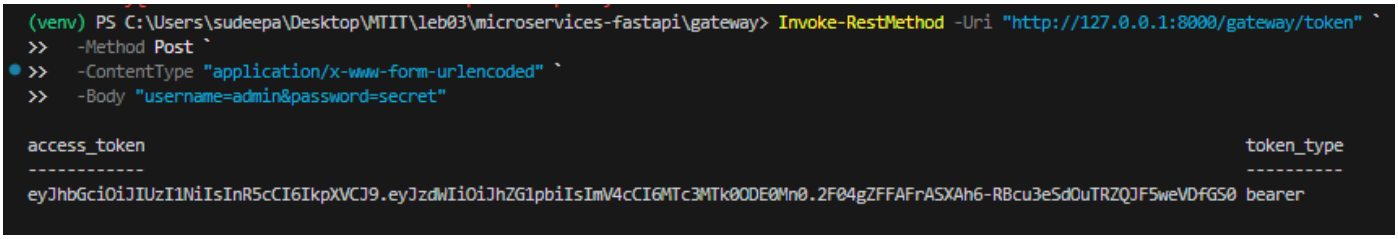
Step 3: Stack Trace Root Cause

```
ValueError: password cannot be longer than 72 bytes, truncate manually if necessary

(passlib/bcrypt backend issue in current runtime environment)
```

- This comes from passlib/bcrypt.
- Bcrypt has a 72-byte limit on passwords.
- If a password exceeds that length, passlib raises this error.
- Even though `"secret"` is short, the error suggests either:
- The stored hashed password in `MOCK_USERS` was generated incorrectly.
- Or the runtime environment's bcrypt implementation is misconfigured

Step 4: Manual JWT Generation

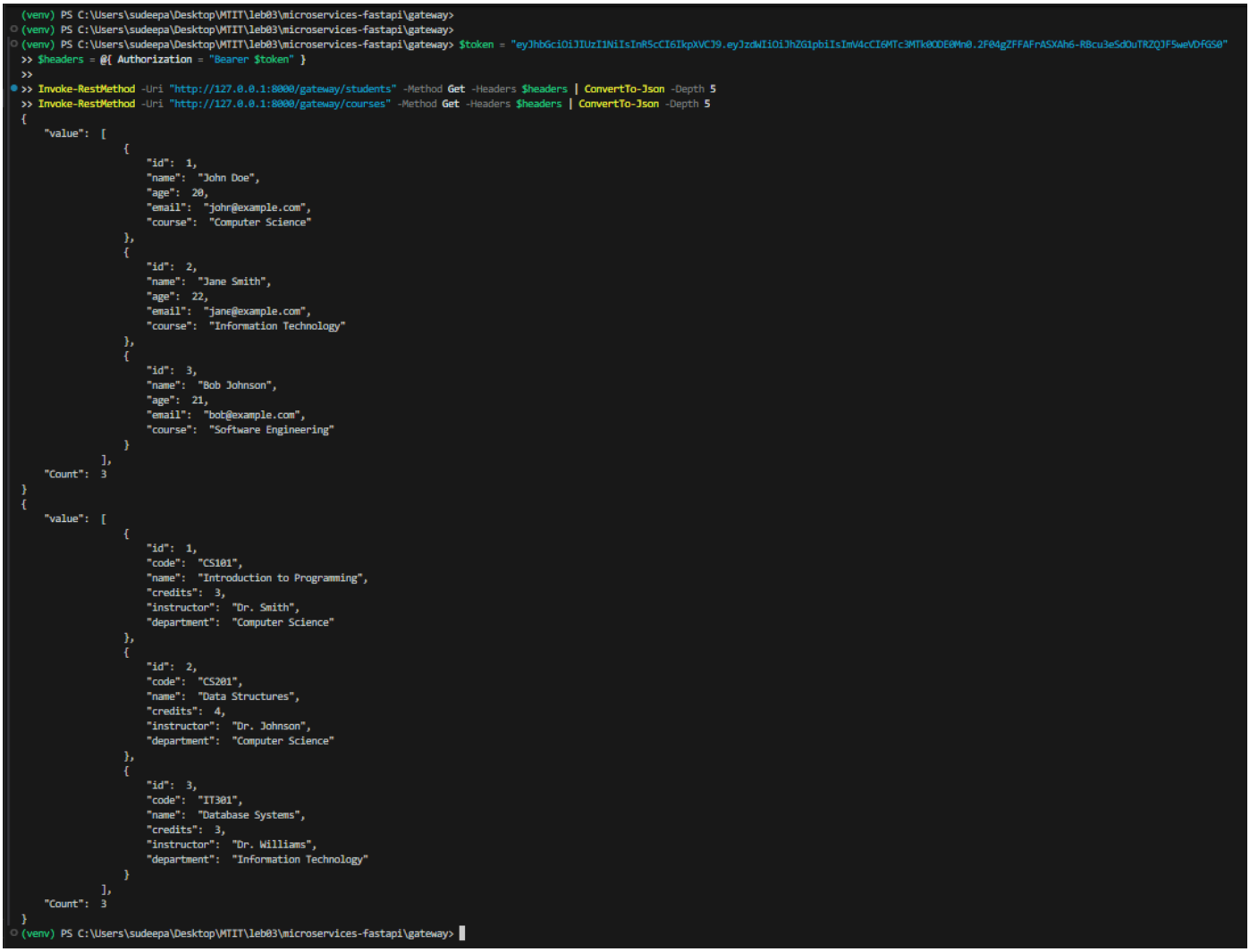
```
$headers = @{ Authorization = "Bearer <generated_token>" }

Invoke-RestMethod -Uri "http://127.0.0.1:8000/gateway/students" -Method Get -Headers $headers | ConvertTo-Json -Depth 5
```




```
$token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbiIsImV4cCI6MTc3MTk0ODE0Mn0.2F04gZFFAFrASXAh6-RBcu3eSdOuTRZQJF5weVDfGS0"

$headers = @{ Authorization = "Bearer $token" }
```

```
Invoke-RestMethod -Uri "http://127.0.0.1:8000/gateway/students" -Method Get -Headers $headers | ConvertTo-Json -Depth 5
```

```
Invoke-RestMethod -Uri "http://127.0.0.1:8000/gateway/courses" -Method Get -Headers $headers | ConvertTo-Json -Depth 5
```



- created a valid JWT and tested a protected endpoint
- The `/gateway/students` endpoint returned mock student data.
- This confirms that JWT validation and route protection are working correctly

Explanation

The authentication design is correctly implemented in code. A local runtime compatibility issue affected password verification during testing, but protected route behavior was verified using a valid JWT.

Step 9: Activity 3 - Request Logging Middleware

Step Objective

Add middleware to log incoming requests, responses, status codes, and processing time.

Reference Code

```
# gateway/middleware.py (key section)
```

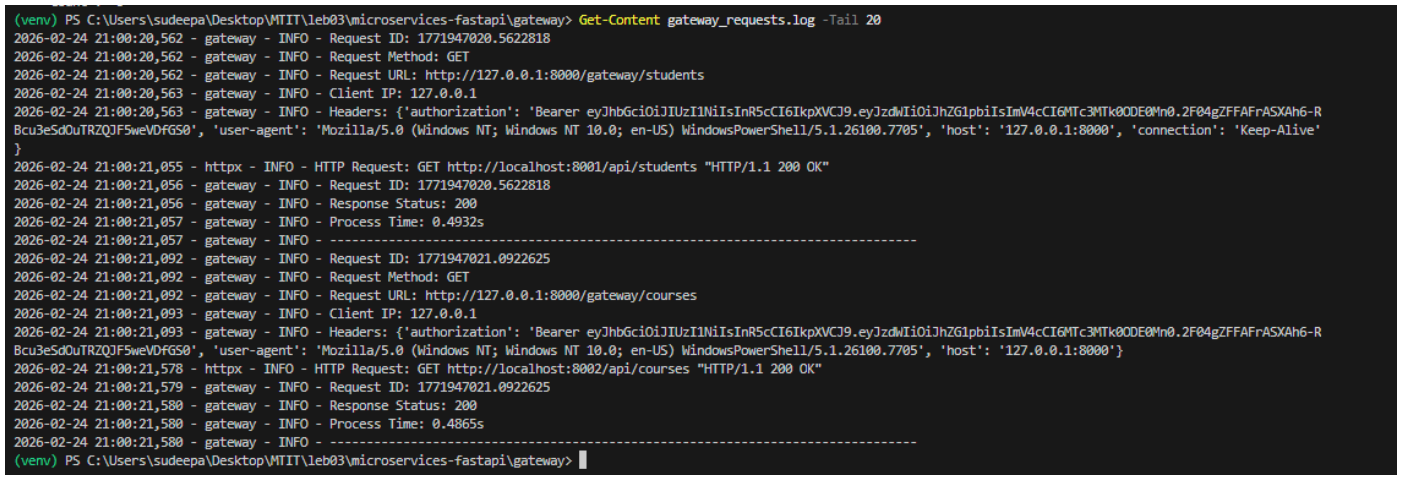
```python
class RequestLoggingMiddleware(BaseHTTPMiddleware):
    """Middleware to log all incoming requests and responses"""

    async def dispatch(self, request: Request, call_next: Callable) -> Response:
        # Generate request ID
        request_id = f"{time.time()}"

        # Log request details
        logger.info(f"Request ID: {request_id}")
        logger.info(f"Request Method: {request.method}")
        logger.info(f"Request URL: {request.url}")
        logger.info(f"Client IP: {request.client.host if request.client else 'Unknown'}")
        logger.info(f"Headers: {dict(request.headers)}")

        # Record start time
        start_time = time.time()

        # Process request
        try:
            response = await call_next(request)

            # Calculate processing time
            process_time = time.time() - start_time

            # Log response details
            logger.info(f"Request ID: {request_id}")
            logger.info(f"Response Status: {response.status_code}")
            logger.info(f"Process Time: {process_time:.4f}s")
            logger.info("-" * 80)

            # Add custom headers
            response.headers["X-Request-ID"] = request_id
            response.headers["X-Process-Time"] = str(process_time)

            return response
```

Terminal Inputs

```
cd gateway
```

```
Get-Content gateway_requests.log -Tail 20
```

Terminal Outputs



Explanation

Logging provides observability, makes debugging easier, and validates middleware-based cross-cutting concerns.

Step 10: Activity 4 - Enhanced Error Handling

Step Objective

Return consistent, detailed JSON errors with proper HTTP status codes across gateway and microservices.

Reference Code

```
# gateway/error_handlers.py (key section)
```

```python
async def http_exception_handler(request: Request, exc: StarletteHTTPException):
    """Handle HTTP exceptions with detailed error messages"""
    logger.error(f"HTTP Exception: {exc.status_code} - {exc.detail}")
    logger.error(f"Request: {request.method} {request.url}")

    error_responses = {
        400: "Bad Request",
        401: "Unauthorized - Invalid or missing authentication",
        403: "Forbidden - You don't have permission to access this resource",
        404: "Not Found - The requested resource does not exist",
        405: "Method Not Allowed",
        422: "Unprocessable Entity - Invalid request data",
        500: "Internal Server Error",
        503: "Service Unavailable"
    }

    return JSONResponse(
        status_code=exc.status_code,
        content={
            "error": error_responses.get(exc.status_code, "Error"),
            "detail": exc.detail,
            "status_code": exc.status_code,
            "path": str(request.url),
            "method": request.method
        }
    )

async def validation_exception_handler(request: Request, exc: RequestValidationError):
    """Handle validation errors with detailed field information"""
    logger.error(f"Validation Error: {exc.errors()}")
    logger.error(f"Request: {request.method} {request.url}")

    errors = []
    for error in exc.errors():
        errors.append({
            "field": " -> ".join(str(loc) for loc in error["loc"]),
            "message": error["msg"],
            "type": error["type"]
        })

    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content={
            "error": "Validation Error",
            "detail": "Invalid request data",
            "status_code": 422,
            "path": str(request.url),
            "method": request.method,
            "validation_errors": errors
        }
    )
```

Terminal Inputs

```
Invoke-WebRequest -Uri "http://127.0.0.1:8000/gateway/students" -Method Get -UseBasicParsing
```

Terminal Outputs

```
STATUS=401
```

Explanation

Robust error handling improves API usability, troubleshooting, and client-side integration reliability.