팀2_문제생성_구현가이드

팀2: 실습 문제 생성 구현 가이드

프로젝트: EduMentor AI
담당: 팀2 (2명)
기간: 2주
목표: 학습자료 기반 난이도별(초급/중급/고급) 실습 문제 자동 생성

📋 팀2 역할 분담

개발자	담당 기능	예상 시간
개발자 C	난이도 분석, 학습 내용 추출, LangGraph Agent 구조	5일
개발자 D	문제 생성 로직, 검증, 답안 및 힌트 생성	5일

🎯 핵심 요구사항

요구사항	목표	검증 방법
난이도 구분	초급/중급/고급 3단계	문제 난이도 점수 검증
문제 유형	코딩, 서술형	문제 타입 필드 확인
자동 답안	모든 문제에 정답 포함	답안 누락 체크
힌트 제공	문제당 2개 이상 힌트	힌트 개수 검증
학습자료 기반	ChromaDB에서 내용 추출	출처 페이지 명시
LangGraph	Agent 워크플로우	다이어그램 작성

Week 1: 기반 구축

Day 1: 환경 설정 및 프로젝트 구조

개발자 C + D (공동 작업)

1.1 Python 프로젝트 생성

```
# 팀2 프로젝트 구조 (python-server/paper_problem/)
paper_problem/
├── __init__.py            # 패키지 초기화
├── models.py              # Pydantic 모델
├── workflow.py            # LangGraph 워크플로우
├── api.py                 # FastAPI 엔드포인트
├── generators/
│   ├── __init__.py
│   ├── beginner.py        # 초급 문제 생성
│   ├── intermediate.py    # 중급 문제 생성
│   └── advanced.py        # 고급 문제 생성
├── validators/
│   ├── __init__.py
│   └── problem_validator.py
└── utils/
    ├── __init__.py
    └── content_analyzer.py  # 문제생성 전용 분석

# 공통 모듈 사용 (python-server/shared/)
shared/
├── __init__.py
```

```
├── chroma_client.py       # ChromaDB 클라이언트 (팀1+팀2 공유)
└── upstage_client.py      # Upstage API 클라이언트 (팀1+팀2 공유)
```

주의: `upstage_client.py`와 `chroma_client.py`는 `shared/` 디렉터리에 있으며, 팀1과 팀2가 공유합니다.

1.2 의존성 설치

```
# requirements.txt (Python 3.12.12 호환)

# LangChain & LangGraph
langchain==0.3.7
langgraph==0.2.45
langsmith==0.1.140
langchain-community==0.3.7
langchain-text-splitters==0.3.2

# Upstage API
upstage==0.10.0
langchain-upstage==0.2.1

# Vector DB
chromadb==0.5.18

# FastAPI
fastapi==0.115.4
uvicorn[standard]==0.32.0
pydantic==2.9.2
pydantic-settings==2.6.1
python-multipart==0.0.12

# PDF/PPT 처리
PyMuPDF==1.24.13
python-pptx==1.0.2

# 유틸리티
python-dotenv==1.0.1
aiohttp==3.10.10
```

```
# 팀1과 동일한 requirements.txt 사용
pip install -r requirements.txt
```

1.3 init.py 작성

```
# paper_problem/__init__.py
"""
팀2: 문제 생성 모듈
난이도별 실습 문제 자동 생성 시스템을 제공합니다.
"""

from paper_problem.api import router as problem_router
from paper_problem.workflow import problem_workflow

__all__ = ['problem_router', 'problem_workflow']
```

```
# paper_problem/generators/__init__.py
"""문제 생성기 모듈"""

from paper_problem.generators.beginner import beginner_generator
from paper_problem.generators.intermediate import intermediate_generator
from paper_problem.generators.advanced import advanced_generator

__all__ = ['beginner_generator', 'intermediate_generator', 'advanced_generator']
```

```
# paper_problem/validators/__init__.py
"""문제 검증 모듈"""

from paper_problem.validators.problem_validator import problem_validator
```

```
__all__ = ['problem_validator']
```

```
# paper_problem/utils/__init__.py
"""문제생성 전용 유틸리티 모듈"""

from paper_problem.utils.content_analyzer import content_analyzer

__all__ = ['content_analyzer']
```

1.4 Pydantic 모델 정의

```
# paper_problem/models.py
from pydantic import BaseModel
from typing import List, Dict, Literal, Optional

class Problem(BaseModel):
    """문제 모델"""
    question: str
    answer: str
    hints: List[str]
    difficulty_score: int  # 1-10
    problem_type: Literal["CODING", "SHORT_ANSWER", "MULTIPLE_CHOICE"]
    test_cases: Optional[List[Dict]] = []
    source_pages: List[int] = []

class ProblemRequest(BaseModel):
    """문제 생성 요청"""
    material_id: int
    difficulty: Literal["BEGINNER", "INTERMEDIATE", "ADVANCED"]
    problem_count: int = 3
    problem_type: Optional[str] = None  # 특정 유형 요청

class ProblemResponse(BaseModel):
    """문제 생성 응답"""
    problems: List[Problem]
    difficulty: str
    generated_count: int
    rejected_count: int
```

Day 2-3: 학습 내용 분석 및 추출 (개발자 C)

2.1 학습 내용 분석기

```
# paper_problem/utils/content_analyzer.py
from typing import List, Dict
from shared.chroma_client import chroma_client
import logging

logger = logging.getLogger(__name__)

class ContentAnalyzer:
    """학습자료 분석 및 핵심 개념 추출"""

    async def analyze_material(
        self,
        material_id: int,
        difficulty: str
    ) -> Dict:
        """난이도에 맞는 학습 내용 추출"""

        # 난이도별 검색 전략
        search_strategies = {
            "BEGINNER": {
                "keywords": ["기본", "개념", "정의", "예제", "소개"],
                "k": 5,
                "focus": "fundamental"
```

```
            },
            "INTERMEDIATE": {
                "keywords": ["실습", "구현", "활용", "응용", "예제"],
                "k": 7,
                "focus": "practical"
            },
            "ADVANCED": {
                "keywords": ["심화", "최적화", "설계", "복잡한", "고급"],
                "k": 10,
                "focus": "advanced"
            }
        }

        strategy = search_strategies[difficulty]

        # 키워드별 검색
        all_docs = []
        seen_ids = set()

        for keyword in strategy["keywords"]:
            results = chroma_client.search(
                collection_name="learning_materials",
                query_texts=[keyword],
                n_results=strategy["k"],
                filter_dict={"material_id": material_id}
            )

            # 중복 제거하며 수집
            for i, doc_id in enumerate(results.get('ids', [[]])[0]):
                if doc_id not in seen_ids:
                    seen_ids.add(doc_id)
                    all_docs.append({
                        'content': results['documents'][0][i],
                        'page': results['metadatas'][0][i]['page'],
                        'type': results['metadatas'][0][i]['type']
                    })

        logger.info(f"Extracted {len(all_docs)} content blocks for {difficulty}")

        return {
            "documents": all_docs,
            "strategy": strategy,
            "total_count": len(all_docs)
        }

    def extract_key_concepts(self, documents: List[Dict]) -> List[str]:
        """핵심 개념 추출"""
        # 간단한 키워드 추출 (실제로는 NER 등 사용 가능)
        concepts = set()

        for doc in documents:
            content = doc['content']
            # 대문자로 시작하는 단어 추출 (클래스명, 개념명 등)
            words = content.split()
            for word in words:
                if word[0].isupper() and len(word) > 3:
                    concepts.add(word)

        return list(concepts)[:10]  # 상위 10개

content_analyzer = ContentAnalyzer()
```

2.2 난이도별 컨텍스트 구성

```
# paper_problem/utils/context_builder.py
from typing import List, Dict

class ContextBuilder:
    """문제 생성을 위한 컨텍스트 구성"""

    def build_context(
```

```
        self,
        documents: List[Dict],
        difficulty: str,
        max_tokens: int = 3000
    ) -> str:
        """난이도에 맞는 컨텍스트 구성"""

        context_parts = []
        current_tokens = 0

        for doc in documents:
            # 대략적인 토큰 계산 (4 chars ≈ 1 token)
            doc_tokens = len(doc['content']) // 4

            if current_tokens + doc_tokens > max_tokens:
                break

            context_parts.append(
                f"[페이지 {doc['page']}]\n{doc['content']}"
            )
            current_tokens += doc_tokens

        context = "\n\n---\n\n".join(context_parts)
        return context

context_builder = ContextBuilder()
```

Day 4-5: 문제 생성 로직 (개발자 D)

3.1 초급 문제 생성기

```
# paper_problem/generators/beginner.py
from langchain_upstage import ChatUpstage
from langchain.schema import HumanMessage
from typing import List
from paper_problem.models import Problem
from shared.upstage_client import upstage_client
import json
import logging

logger = logging.getLogger(__name__)

class BeginnerProblemGenerator:
    """초급 실습 문제 생성"""

    def __init__(self):
        self.llm = upstage_client.get_chat_model(
            model="solar-1-mini-chat",
            temperature=0.7
        )

    async def generate(
        self,
        context: str,
        count: int = 3
    ) -> List[Problem]:
        """초급 문제 생성"""

        prompt = f"""다음 학습 내용을 바탕으로 **초급** 실습 문제를 {count}개 생성하세요.

**학습 내용**:
{context}

**초급 문제 요구사항**:
1. 기본 개념 이해 확인
2. 단순한 코드 작성 (5-10줄)
3. 명확한 정답이 있는 문제
```

```
4. 실제 학습 내용에서 다룬 내용만 사용

**출력 형식** (JSON):
[
    {{
        "question": "문제 내용 (구체적으로)",
        "answer": "정답 또는 예시 코드",
        "hints": ["힌트1", "힌트2"],
        "difficulty_score": 1-3 (1이 가장 쉬움),
        "problem_type": "CODING" 또는 "SHORT_ANSWER",
        "test_cases": [{{"input": "...", "expected": "..."}}]  // CODING인 경우만
    }},
    ...
]

**중요**: 반드시 유효한 JSON 배열로 출력하세요."""

        response = await self.llm.ainvoke([HumanMessage(content=prompt)])

        try:
            problems_data = json.loads(response.content)
            problems = [Problem(**p) for p in problems_data]
            logger.info(f"Generated {len(problems)} beginner problems")
            return problems
        except json.JSONDecodeError as e:
            logger.error(f"Failed to parse JSON: {e}")
            logger.error(f"Response: {response.content}")
            return []

beginner_generator = BeginnerProblemGenerator()
```

3.2 중급 문제 생성기

```
# paper_problem/generators/intermediate.py
from langchain_upstage import ChatUpstage
from langchain.schema import HumanMessage
from typing import List
from paper_problem.models import Problem
from shared.upstage_client import upstage_client
import json
import logging

logger = logging.getLogger(__name__)

class IntermediateProblemGenerator:
    """중급 실습 문제 생성"""

    def __init__(self):
        self.llm = upstage_client.get_chat_model(
            model="solar-1-mini-chat",
            temperature=0.8  # 더 다양한 문제 생성
        )

    async def generate(
        self,
        context: str,
        count: int = 3
    ) -> List[Problem]:
        """중급 문제 생성"""

        prompt = f"""다음 학습 내용을 바탕으로 **중급** 실습 문제를 {count}개 생성하세요.

**학습 내용**:
{context}

**중급 문제 요구사항**:
1. 여러 개념을 결합
2. 실무 시나리오 기반
3. 코드 작성 (20-30줄)
4. 테스트 케이스 포함
5. 학습 내용에서 다룬 개념만 사용
```

```
**출력 형식** (JSON):
[
    {{
        "question": "실무 상황을 반영한 문제 내용",
        "answer": "정답 코드 또는 설명",
        "hints": ["힌트1 (첫 단계)", "힌트2 (핵심 개념)"],
        "difficulty_score": 4-6,
        "problem_type": "CODING",
        "test_cases": [
            {{"input": "...", "expected": "..."}},
            {{"input": "...", "expected": "..."}}
        ]
    }},
    ...
]

**중요**: 반드시 유효한 JSON 배열로 출력하세요."""

        response = await self.llm.ainvoke([HumanMessage(content=prompt)])

        try:
            problems_data = json.loads(response.content)
            problems = [Problem(**p) for p in problems_data]
            logger.info(f"Generated {len(problems)} intermediate problems")
            return problems
        except json.JSONDecodeError as e:
            logger.error(f"Failed to parse JSON: {e}")
            return []

intermediate_generator = IntermediateProblemGenerator()
```

3.3 고급 문제 생성기

```
# paper_problem/generators/advanced.py
from langchain_upstage import ChatUpstage
from langchain.schema import HumanMessage
from typing import List
from paper_problem.models import Problem
from shared.upstage_client import upstage_client
import json
import logging

logger = logging.getLogger(__name__)

class AdvancedProblemGenerator:
    """고급 실습 문제 생성"""

    def __init__(self):
        self.llm = upstage_client.get_chat_model(
            model="solar-1-mini-chat",
            temperature=0.9  # 창의적인 문제 생성
        )

    async def generate(
        self,
        context: str,
        count: int = 3
    ) -> List[Problem]:
        """고급 문제 생성"""

        prompt = f"""다음 학습 내용을 바탕으로 **고급** 실습 문제를 {count}개 생성하세요.

**학습 내용**:
{context}

**고급 문제 요구사항**:
1. 복잡한 실무 시나리오
2. 최적화 또는 설계 문제
3. 여러 파일/클래스 구성 필요
4. 엣지 케이스 고려
```

```
5. 성능 및 확장성 고려

**출력 형식** (JSON):
[
    {{
        "question": "복잡한 실무 시나리오 문제",
        "answer": "정답 코드 또는 설계 설명",
        "hints": ["힌트1 (아키텍처 힌트)", "힌트2 (최적화 힌트)"],
        "difficulty_score": 7-10,
        "problem_type": "CODING",
        "test_cases": [
            {{"input": "일반 케이스", "expected": "..."}},
            {{"input": "엣지 케이스", "expected": "..."}},
            {{"input": "성능 테스트", "expected": "..."}}
        ]
    }},
    ...
]

**중요**: 반드시 유효한 JSON 배열로 출력하세요."""

        response = await self.llm.ainvoke([HumanMessage(content=prompt)])

        try:
            problems_data = json.loads(response.content)
            problems = [Problem(**p) for p in problems_data]
            logger.info(f"Generated {len(problems)} advanced problems")
            return problems
        except json.JSONDecodeError as e:
            logger.error(f"Failed to parse JSON: {e}")
            return []

advanced_generator = AdvancedProblemGenerator()
```

Week 2: 검증 및 완성

Day 6-7: 문제 검증 및 LangGraph 워크플로우

4.1 문제 검증기

```
# paper_problem/validators/problem_validator.py
from paper_problem.models import Problem
from typing import List, Tuple
import logging

logger = logging.getLogger(__name__)

class ProblemValidator:
    """생성된 문제 검증"""

    def validate(self, problem: Problem, difficulty: str) -> Tuple[bool, str]:
        """문제 유효성 검증"""

        # 1. 필수 필드 검증
        if not problem.question or len(problem.question) < 50:
            return False, "Question too short (minimum 50 characters)"

        if not problem.answer or len(problem.answer) < 20:
            return False, "Answer too short (minimum 20 characters)"

        if len(problem.hints) < 2:
            return False, "Need at least 2 hints"

        # 2. 난이도 점수 검증
        difficulty_ranges = {
            "BEGINNER": (1, 3),
            "INTERMEDIATE": (4, 6),
            "ADVANCED": (7, 10)
```

```
        }

        min_score, max_score = difficulty_ranges[difficulty]
        if not (min_score <= problem.difficulty_score <= max_score):
            return False, f"Difficulty score must be between {min_score}-{max_score}"

        # 3. 코딩 문제 검증
        if problem.problem_type == "CODING":
            if not problem.test_cases or len(problem.test_cases) == 0:
                return False, "CODING problems need test cases"

            # 테스트 케이스 구조 검증
            for tc in problem.test_cases:
                if 'input' not in tc or 'expected' not in tc:
                    return False, "Test case needs 'input' and 'expected' fields"

        # 4. 답변 품질 검증
        if "TODO" in problem.answer or "..." in problem.answer:
            return False, "Answer contains placeholder text"

        return True, "Valid"

    def filter_valid_problems(
        self,
        problems: List[Problem],
        difficulty: str
    ) -> Tuple[List[Problem], List[str]]:
        """유효한 문제만 필터링"""

        valid_problems = []
        rejection_reasons = []

        for problem in problems:
            is_valid, reason = self.validate(problem, difficulty)

            if is_valid:
                valid_problems.append(problem)
                logger.info(f"✅ Valid problem: {problem.question[:50]}...")
            else:
                rejection_reasons.append(reason)
                logger.warning(f"❌ Rejected: {reason}")

        return valid_problems, rejection_reasons

problem_validator = ProblemValidator()
```

4.2 LangGraph 워크플로우 구성

```
# paper_problem/workflow.py
from langgraph.graph import StateGraph, START, END
from typing import TypedDict, List, Dict
from paper_problem.models import Problem
from paper_problem.utils.content_analyzer import content_analyzer
from paper_problem.generators.beginner import beginner_generator
from paper_problem.generators.intermediate import intermediate_generator
from paper_problem.generators.advanced import advanced_generator
from paper_problem.validators.problem_validator import problem_validator
import logging

logger = logging.getLogger(__name__)

class ProblemState(TypedDict):
    material_id: int
    difficulty: str
    problem_count: int
    learning_content: Dict
    context: str
    generated_problems: List[Problem]
    validated_problems: List[Problem]
    rejection_reasons: List[str]
```

```python
# 노드 1: 학습 내용 분석
async def analyze_content_node(state: ProblemState) -> dict:
    """학습자료에서 핵심 내용 추출"""
    material_id = state["material_id"]
    difficulty = state["difficulty"]

    logger.info(f"Analyzing content for {difficulty} problems")

    # 학습 내용 분석
    analysis = await content_analyzer.analyze_material(
        material_id=material_id,
        difficulty=difficulty
    )

    return {"learning_content": analysis}

# 노드 2: 컨텍스트 구성
async def build_context_node(state: ProblemState) -> dict:
    """문제 생성을 위한 컨텍스트 구성"""
    learning_content = state["learning_content"]
    difficulty = state["difficulty"]

    documents = learning_content["documents"]

    # 컨텍스트 구성
    context = context_builder.build_context(
        documents=documents,
        difficulty=difficulty
    )

    logger.info(f"Built context: {len(context)} characters")

    return {"context": context}

# 노드 3: 문제 생성
async def generate_problems_node(state: ProblemState) -> dict:
    """난이도별 문제 생성"""
    difficulty = state["difficulty"]
    context = state["context"]
    problem_count = state["problem_count"]

    logger.info(f"Generating {problem_count} {difficulty} problems")

    # 난이도별 생성기 선택
    if difficulty == "BEGINNER":
        problems = await beginner_generator.generate(context, problem_count)
    elif difficulty == "INTERMEDIATE":
        problems = await intermediate_generator.generate(context, problem_count)
    else:  # ADVANCED
        problems = await advanced_generator.generate(context, problem_count)

    return {"generated_problems": problems}

# 노드 4: 문제 검증
async def validate_problems_node(state: ProblemState) -> dict:
    """생성된 문제 검증 및 필터링"""
    problems = state["generated_problems"]
    difficulty = state["difficulty"]

    logger.info(f"Validating {len(problems)} problems")

    # 검증
    validated_problems, rejection_reasons = problem_validator.filter_valid_problems(
        problems=problems,
        difficulty=difficulty
    )

    logger.info(f"Validated: {len(validated_problems)}/{len(problems)} problems")

    return {
        "validated_problems": validated_problems,
        "rejection_reasons": rejection_reasons
```

```
    }

# 노드 5: 재생성 판단
def should_regenerate(state: ProblemState) -> str:
    """문제가 부족하면 재생성"""
    validated_count = len(state["validated_problems"])
    required_count = state["problem_count"]

    if validated_count < required_count:
        logger.warning(f"Only {validated_count}/{required_count} problems valid. Regenerating...")
        return "regenerate"
    else:
        return "end"

# 워크플로우 생성
def create_problem_workflow():
    graph = StateGraph(ProblemState)

    # 노드 추가
    graph.add_node("analyze", analyze_content_node)
    graph.add_node("build_context", build_context_node)
    graph.add_node("generate", generate_problems_node)
    graph.add_node("validate", validate_problems_node)

    # 엣지
    graph.add_edge(START, "analyze")
    graph.add_edge("analyze", "build_context")
    graph.add_edge("build_context", "generate")
    graph.add_edge("generate", "validate")

    # 조건부 엣지 (재생성 판단)
    graph.add_conditional_edges(
        "validate",
        should_regenerate,
        {
            "regenerate": "generate",  # 다시 생성
            "end": END
        }
    )

    return graph.compile()

problem_workflow = create_problem_workflow()
```

Day 8-9: API 구현 및 테스트

5.1 FastAPI 엔드포인트

```
# paper_problem/api.py
from fastapi import APIRouter, HTTPException
from paper_problem.models import ProblemRequest, ProblemResponse
from paper_problem.workflow import problem_workflow
import logging

logger = logging.getLogger(__name__)

router = APIRouter(prefix="/problems", tags=["Problems"])

@router.post("/generate", response_model=ProblemResponse)
async def generate_problems(request: ProblemRequest):
    """난이도별 실습 문제 생성"""

    logger.info(
        f"Generating {request.problem_count} {request.difficulty} problems "
        f"for material {request.material_id}"
    )

    try:
```

```
        # LangGraph 워크플로우 실행
        result = await problem_workflow.ainvoke({
            "material_id": request.material_id,
            "difficulty": request.difficulty,
            "problem_count": request.problem_count
        })

        validated_problems = result["validated_problems"]
        rejection_reasons = result["rejection_reasons"]

        if not validated_problems:
            raise HTTPException(
                status_code=500,
                detail="Failed to generate valid problems"
            )

        return ProblemResponse(
            problems=validated_problems,
            difficulty=request.difficulty,
            generated_count=len(validated_problems),
            rejected_count=len(rejection_reasons)
        )
```

```
    except Exception as e:
        logger.error(f"Error generating problems: {e}")
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/difficulties")
async def get_difficulty_info():
    """난이도별 정보 조회"""
    return {
        "BEGINNER": {
            "score_range": "1-3",
            "description": "기본 개념 이해 확인",
            "example": "JPA Entity 클래스 작성하기"
        },
        "INTERMEDIATE": {
            "score_range": "4-6",
            "description": "실무 시나리오 기반 실습",
            "example": "게시판 CRUD API 구현하기"
        },
        "ADVANCED": {
            "score_range": "7-10",
            "description": "복잡한 설계 및 최적화",
            "example": "대용량 트래픽을 위한 캐싱 전략 설계"
        }
    }
```

5.2 통합 테스트

```
# tests/test_problem_generation.py
import asyncio
from paper_problem.workflow import problem_workflow

async def test_problem_generation():
    """문제 생성 전체 테스트"""

    difficulties = ["BEGINNER", "INTERMEDIATE", "ADVANCED"]

    for difficulty in difficulties:
        print(f"\n{'='*50}")
        print(f"Testing {difficulty} problem generation")
        print('='*50)

        result = await problem_workflow.ainvoke({
            "material_id": 1,
            "difficulty": difficulty,
            "problem_count": 3
        })

        problems = result["validated_problems"]
```

```
        print(f"\n✅ Generated {len(problems)} valid problems")

        for i, problem in enumerate(problems, 1):
            print(f"\n[문제 {i}]")
            print(f"난이도 점수: {problem.difficulty_score}")
            print(f"유형: {problem.problem_type}")
            print(f"질문: {problem.question[:100]}...")
            print(f"힌트 개수: {len(problem.hints)}")
            print(f"테스트 케이스: {len(problem.test_cases)}")

        print(f"\n❌ Rejected: {len(result['rejection_reasons'])} problems")
        for reason in result['rejection_reasons']:
            print(f"  - {reason}")

if __name__ == "__main__":
    asyncio.run(test_problem_generation())
```

```
# 실행
python tests/test_problem_generation.py
```

Day 10: 문서화 및 완성

6.1 메인 서버 파일

```
# main.py (Python 서버 루트)
from fastapi import FastAPI
from paper_qa.api import router as qa_router
from paper_problem.api import router as problem_router
import logging

# 로깅 설정
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

app = FastAPI(
    title="EduMentor AI Engine",
    description="QA 시스템 + 문제 생성 통합 API",
    version="1.0.0"
)

# 팀1 QA 라우터 등록
app.include_router(qa_router, prefix="/qa", tags=["QA"])

# 팀2 문제 생성 라우터 등록
app.include_router(problem_router, prefix="/problems", tags=["Problems"])

@app.get("/")
async def root():
    return {
        "service": "EduMentor AI Engine",
        "version": "1.0.0",
        "endpoints": {
            "qa": "/qa",
            "problems": "/problems",
            "docs": "/docs"
        }
    }

@app.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "services": ["qa", "problems"]
    }
```

```python
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

6.2 README 작성

```
# 팀2: 실습 문제 생성 시스템

## 개요

학습자료를 기반으로 난이도별(초급/중급/고급) 실습 문제를 자동 생성합니다.

## 설치

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

실행

```bash
# 팀1 서버가 먼저 실행되어 있어야 함 (ChromaDB 연동)
python main.py
```

서버: http://localhost:8001

API 사용법

1. 초급 문제 생성

```bash
curl -X POST http://localhost:8001/problems/generate \
  -H "Content-Type: application/json" \
  -d '{
    "material_id": 1,
    "difficulty": "BEGINNER",
    "problem_count": 3
  }'
```

2. 중급 문제 생성

```bash
curl -X POST http://localhost:8001/problems/generate \
  -H "Content-Type: application/json" \
  -d '{
    "material_id": 1,
    "difficulty": "INTERMEDIATE",
    "problem_count": 3
  }'
```

3. 고급 문제 생성

```bash
curl -X POST http://localhost:8001/problems/generate \
  -H "Content-Type: application/json" \
  -d '{
    "material_id": 1,
    "difficulty": "ADVANCED",
    "problem_count": 3
  }'
```

문제 구조

```
{
  "question": "문제 내용",
  "answer": "정답",
  "hints": ["힌트1", "힌트2"],
  "difficulty_score": 5,
  "problem_type": "CODING",
  "test_cases": [
```
```

```
    {"input": "입력", "expected": "기대 출력"}
  ]
}
```

난이도 기준

난이도	점수	설명
BEGINNER	1-3	기본 개념, 간단한 코드 (5-10줄)
INTERMEDIATE	4-6	실무 시나리오, 중간 코드 (20-30줄)
ADVANCED	7-10	복잡한 설계, 최적화 고려

검증 규칙

- 질문 최소 50자
- 답변 최소 20자
- 힌트 2개 이상
- 코딩 문제는 테스트 케이스 필수

📊 팀2 완성 체크리스트

핵심 기능

- [ ] 학습 내용 분석 (ChromaDB 검색)
- [ ] 난이도별 문제 생성 (초급/중급/고급)
- [ ] 문제 검증 시스템
- [ ] LangGraph 워크플로우
- [ ] 재생성 로직 (검증 실패 시)

문제 유형

- [ ] 코딩 문제 (CODING)
- [ ] 서술형 문제 (SHORT_ANSWER)
- [ ] 테스트 케이스 생성

API 엔드포인트

- [ ] POST `/problems/generate` - 문제 생성
- [ ] GET `/problems/difficulties` - 난이도 정보

검증 항목

- [ ] 질문 길이 (최소 50자)
- [ ] 답변 길이 (최소 20자)
- [ ] 힌트 개수 (최소 2개)
- [ ] 난이도 점수 검증
- [ ] 테스트 케이스 검증

테스트

- [ ] 난이도별 문제 생성 테스트
- [ ] 검증 로직 테스트
- [ ] 재생성 시나리오 테스트

문서화

- [ ] API 문서 (FastAPI Swagger)
- [ ] README.md
- [ ] 코드 주석

🚀 다음 단계

팀2 완료 후:

1. **팀1과 통합**: Spring Boot 연동
2. **전체 시스템 테스트**: QA + 문제 생성 통합
3. **UI 개발**: 문제 표시 및 제출 화면
4. **배포 준비**: Docker 컨테이너화

작성일: 2025-10-28
담당: 팀2 (개발자 C, D)