

개별연구 보고서

학생 개별 작성용

수행 학기	2022 학년도 2 학기			
교과목 정보	교과목명	CS 웹 IDE 기반 문제은행 개선연구	분반	1 분반
교과담당교수	소속	컴퓨터공학전공	교수명	
	이름		학번	
	학과		전화번호	
	학년		이메일	
학생정보				
요약 보고서				
작품명 (프로젝트명)	웹 기반 문제은행 시스템 구현			
	1. 연구·개발 동기/목적/ 필요성 및 개발 목표			
최종 결과물 소개	현재 다양한 사이트에서 웹 기반 문제 은행 및 채점 시스템을 운영하고 있다. 이번 프로젝트에서는 해당 시스템이 어떻게 동작할 수 있을지에 대해 연구하고 이를 간단하게 구현함으로써 웹 기반 프로그래밍 IDE의 동작 방식에 대해 이해하고, 추가적인 개선 방향에 대해 논해보고자 한다.			
	2. 웹 기반 문제 풀이 및 문제 생성 시스템 구현 문자열 비교 기반 채점 시스템 구현 문제, 테스트케이스, 초기 제공 코드 등 웹 기반 문제 풀이와 관련된 요소에 대한 데이터베이스 설계			
3. 프로젝트 추진 내용	문제 페이지를 통해 온라인에서 직접 코딩이 가능하며, 서버에 코드를 제출하여 채점을 진행할 수 있다. 채점 결과를 보고 사용자는 해당 문제의 성공 여부를 알 수 있다. 사용자가 서버로 코드를 제출하는 경우, 서버에서는 해당 언어에 대응되는 _ScoringManager 을 통해 채점을 진행할 수 있다. ScoringManager 은 언어 측면의 확장성을 위해 상속 관계로 구현하였으며, 팩토리 메서드를 통해 일관성 있는 사용이 가능하다. ScoringManager 에서는 대응되는 언어에 대한 동작을 수행한다. 문제와 관련된 모든 실행 과정은 exec 을 통해 프로세스를 생성한 후 진행되며 컴파일, 초기화(코드에 대응되는 파일 생성), 실행, 문자열 기반 결과값 비교, 해제 과정을 일관된 코드로 수행하도록 유도하되, 언어마다 차이가 있는 부분에 대해서는 상속한 클래스에서 값 또는 함수를 넘기는 형식으로 일관되게 처리했다.			
	4. 기대효과 문제 은행 시스템의 구조를 연구하고 이를 실제로 구현함으로써 웹 생태계 전반과 다양한 프레임워크에 대한 이해도를 높일 수 있다. 문자열 기반 값 비교를 통해 모든 언어에 대해 일관된 방식의 채점 시스템을 제시함으로써 새로운 언어에 대한 확장성을 높인다. 웹 환경을 통한 문제 풀이를 수행할 수 있으므로 다른 방식에 비해 접근성이 높고 환경적 제약이 낮다.			

목차

1. 서론	3
2 관련/배경 연구	3
2.1 기존 시스템 분석	3
2.1.1 BOJ	3
2.1.2 프로그래머스	5
2.2 구현 방향 설정	8
2.3 구현 방법 연구	8
2.3.1 유저 프로그램 실행	8
2.3.2 입력	9
2.3.3 출력	10
3. 본론	11
3.1 전체 프로젝트 구조	11
3.2 데이터베이스 설계	12
3.3 백엔드 구현	14
3.3.1 컨트롤러	14
3.3.2 채점 시스템	15
3.3.3 라우터	20
3.4 프론트엔드 구현	21
4. 구현 결과	22
4.1 메인 페이지	22
4.2 문제 목록 페이지	23
4.3 문제 생성 페이지	24
4.4 문제풀이 페이지	26
4.5. 실행 방법	29
5. 결론 및 향후 연구	30

1. 서론

웹 기반 IDE 을 통한 문제 풀이 시스템은 사용자의 환경적 특성에 크게 좌우되지 않으므로 모든 사용자들이 모두 동일한 환경 속에서 문제를 풀 수 있도록 지원한다. 국내의 경우 BOJ¹, 프로그래머스² 등의 웹 사이트가 이러한 서비스를 제공하고 있다.

현재 프로젝트에서는 언급한 웹 사이트들의 채점 시스템이 어떻게 구현되는지에 대해 분석하고, 분석 결과를 기반으로 문제 채점 시스템을 구현함으로써 웹 IDE 기반 문제풀이 서비스의 동작 방식을 이해하고, 현재 프로젝트에서의 개선 방향을 논의해보고자 한다.

2 관련/배경 연구

현재 파트에서는 서론에서 언급한 BOJ, 프로그래머스의 문제 채점 방식에 대해 분석하고, 구현 방향을 설정한다.

2.1 기존 시스템 분석

2.1.1 BOJ

<https://www.acmicpc.net/problem/10430>

The screenshot shows the BOJ problem page for problem 10430. It includes a table with problem statistics, the problem description, input/output specifications, and example input/output. Red boxes and numbered circles (1-5) highlight specific elements: 1. Problem statistics table, 2. Problem description, 3. Input section, 4. Output section, 5. Example input/output section.

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	274218	145099	126751	53.439%

문제

$(A+B)\%C$ 는 $((A\%C) + (B\%C))\%C$ 와 같을까?
 $(A\times B)\%C$ 는 $((A\%C) \times (B\%C))\%C$ 와 같을까?
세 수 A, B, C가 주어졌을 때, 위의 네 가지 값을 구하는 프로그램을 작성하시오.

입력

첫째 줄에 A, B, C가 순서대로 주어진다. ($2 \leq A, B, C \leq 10000$)

출력

첫째 줄에 $(A+B)\%C$, 둘째 줄에 $((A\%C) + (B\%C))\%C$, 셋째 줄에 $(A\times B)\%C$, 넷째 줄에 $((A\%C) \times (B\%C))\%C$ 를 출력한다.

예제 입력 1 복사

```
5 8 4
```

예제 출력 1 복사

```
1
1
0
0
```

그림 1 BOJ 의 문제 포맷

¹ <https://www.acmicpc.net/>

² <https://programmers.co.kr/>

BOJ의 문제 페이지에는 다음과 같은 정보들이 표현된다.

1. 문제 정보: 문제에 대한 제한사항, 제출 및 정답 현황 등 문제에 관련된 정보를 보여준다.
2. 설명: 문제에 대한 설명을 제시한다.
3. 입력: 입력되는 값의 특성을 제시한다.
4. 출력: 출력되는 값의 특성을 제시한다.
5. 예제 입력/출력: 입력/출력 되는 값의 예시를 제시한다.

위 정보들 중 예제 입력과 예제 출력은 다음과 같은 모습으로 제시된다.

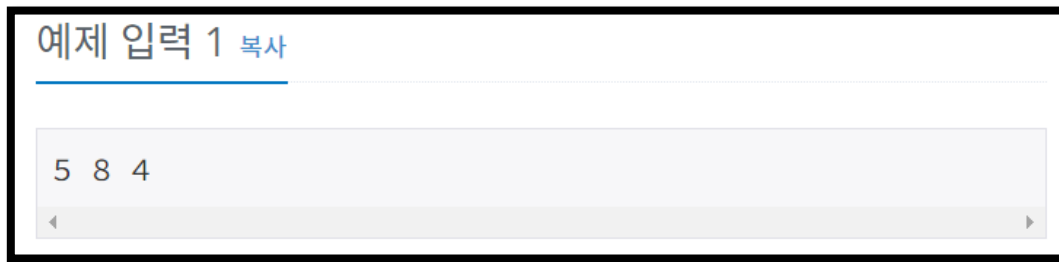


그림 2 예제 입력

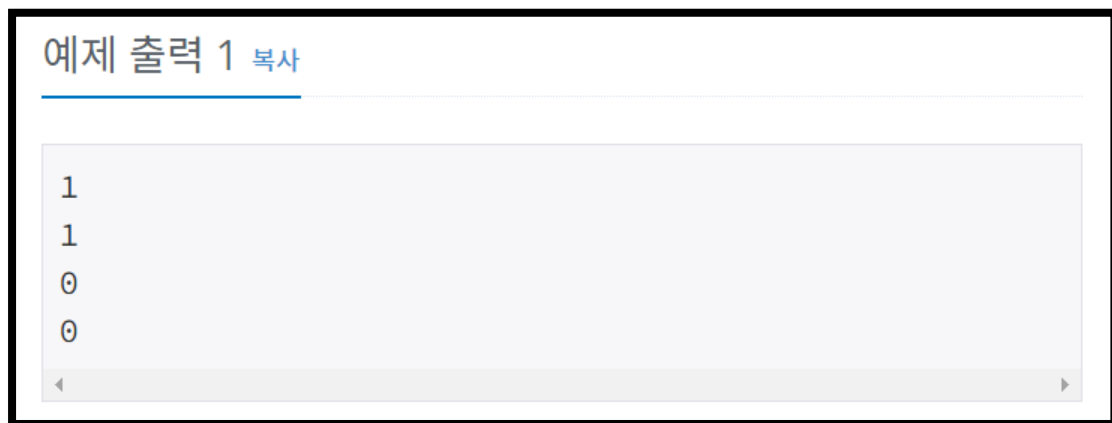


그림 3 예제 출력

그림에서 볼 수 있듯이 BOJ의 입력 및 출력은 문자열 형태로 주어진다. 따라서 사용자가 문제를 푸는 경우 stdin을 통해 입력을 받고 stdout으로 출력하는 코드를 작성할 필요가 있다.

문자열 기반으로 입출력을 설정하는 경우 장점 및 단점은 다음과 같다.

장점

1. 입력 및 출력으로 사용되는 데이터가 특정 언어의 특성 또는 문법에 국한되지 않는 표준 입출력으로 구성되므로 언어에 대한 확장성이 높다. 특정 시점에 새로운 프로그래밍 언어를 지원하더라도 각 문제는 프로그래밍 언어와 독립적이므로 문제와 관련된 어떠한 변동 및 추가 사항 없이 언어가 추가될 수 있다. 즉, 언어와 문제를 독립적으로 취급할 수 있다.
2. 입력의 확장성이 매우 높다. 한 문제에 대한 모든 입력은 표준 입력이므로, 사용자는 자신이 사용하는 언어의 stdin 입력 지원 함수를 이용하여 문자열 형태로 값을 입력 받고 이를 가공하여

문제 내에서 사용하게 된다. 이 경우 입력 값을 가공하는 책임은 사용자에게 있으므로 문제 제작자는 입력 구조를 자유롭게 설정할 수 있다. 이에 의해 문제의 구조에 제한 사항이 적다.

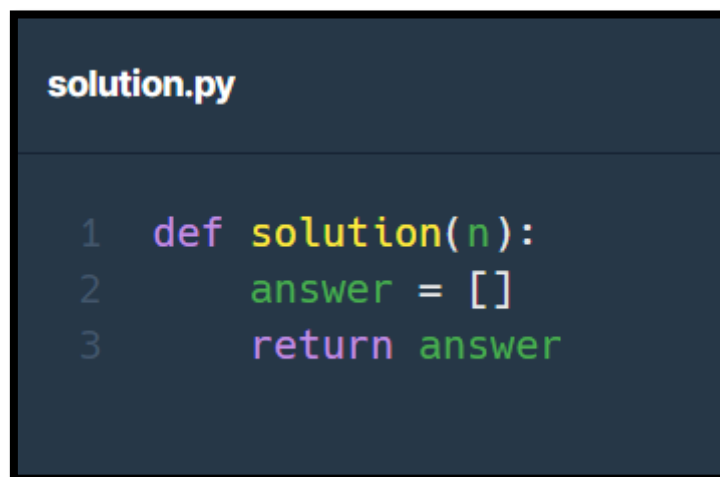
3. 표준 출력을 문자열로 취급하는 경우 테스트케이스의 비교는 문자열 비교로 단순화된다. 이 경우 테스트케이스의 비교가 각 언어 차원에서 수행될 필요 없이 단일 언어로 작성된 채점 시스템에 의해 진행될 수 있으므로 각 문제 또는 언어에 대응되는 채점 시스템을 구현할 필요가 없다.

단점

1. 문제를 해결하는 사용자에게 입력 값을 가공할 책임이 부과된다. 프로그래밍 문제의 목적은 사용자의 문제 해결 능력을 기르는데 있다. 일반적인 경우 입력의 가공은 프로그래밍 문제에서 다루고자 하는 주된 목표가 아니라 부차적인 역할을 수행하게 되는데, 표준 입출력 기반으로 입력 및 출력을 다루는 현재 방식에서는 사용자에게 해당 역할을 부과함으로써 문제 해결 능력과 직접적으로 상관없는 코드를 다뤄야 하는 비효율이 발생한다. 다만 이러한 영역까지도 문제 해결 능력의 일환으로 취급할 수 있다면 큰 문제는 아니다.

2.1.2 프로그래머스

프로그래머스의 문제 페이지³⁴에서 나타나는 정보는 BOJ와 유사하나, 입출력 부분에서 궁극적인 차이가 존재한다. BOJ와 달리 프로그래머스의 입출력은 함수의 파라미터 형식으로 제공된다.



```
solution.py

1  def solution(n):
2      answer = []
3      return answer
```

그림 4 파이썬 초기 코드

위 그림은 프로그래머스 문제 페이지에서 제공하는 초기 코드 중 하나를 제시한 것이다. BOJ와는 달리 n 이라는 파라미터를 통해 입력 값이 제공되며 `return`을 통해 출력 값을 전달한다. 일반적인 프로그래밍 과정에서 함수를 구성하는 과정과 유사하게 코딩이 진행된다는 특징을 가진다.

³ 출처: 프로그래머스 코딩 테스트 연습, <https://school.programmers.co.kr/learn/challenges>

⁴ 이미지 출처, <https://school.programmers.co.kr/learn/courses/30/lessons/120897>

각 언어마다 입력 값의 타입 및 함수를 구성하는데 사용되는 문법 등이 상당히 다르므로 BOJ 의 방식과는 달리 언어와 문제에 대한 초기 코드가 상당히 긴밀하게 연결된다.

```
solution.c
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <stdlib.h>
4
5  int* solution(int n) {
6      // return 값은 malloc 등 동적 할당
7      int* answer = (int*)malloc(1);
8      return answer;
9  }
```

그림 5 C 초기 코드

```
solution.js
1  function solution(n) {
2      var answer = [];
3      return answer;
4  }
```

그림 6 자바스크립트 초기 코드

그림 3, 4, 5 는 동일한 문제에 대한 서로 다른 언어의 초기 코드를 보여준다. 여기서 볼 수 있듯이 각 언어에 대한 초기 코드는 해당 언어의 문법에 맞게 정의되어 있으며, 내부에서 사용되는 자료구조 역시 해당 언어에서 가능한 방법을 기반으로 정의된다. 사용자 입장에서는 입력 및 출력이 미리 정의되어 있으므로 이를 고려할 필요가 없다는 장점이 존재한다.

프로그래머스는 내부적으로 채점이 어떻게 수행되는지에 대해 공개하고 있지는 않다. 하지만 사용자에게 제공되는 초기화 코드가 함수 형태이며, 외부로부터 입력을 받아 결과를 출력하는 구조를 가지고 있다는 점에 있어서 해당 함수를 실행하는 외부의 main 함수가 존재할 것이라고 추론해볼 수 있다. 이러한 추론을 기반으로 한 채점 방식을 간단하게 그림으로 나타내면 다음과 같다.

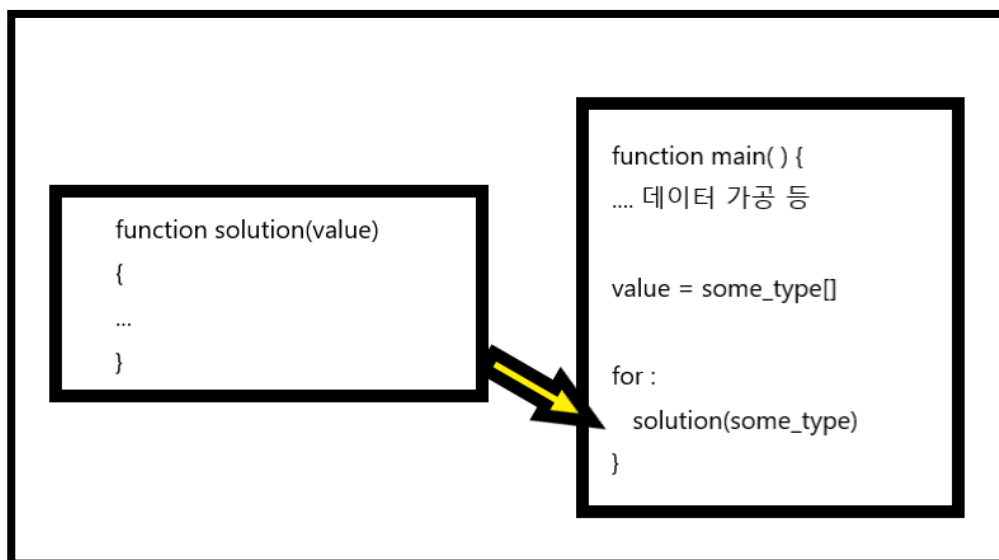


그림 7 프로그래머스 코드의 실행 방식 예상도

1. 사용자가 자신이 작성한 코드를 제출하면 서버 상에서 해당 코드를 담은 파일을 생성한다.
2. main 함수에서 solution 함수를 테스트 케이스의 개수만큼 for 문을 통해 호출한다.

3. solution 함수는 호출될 때마다 특정한 입력에 대한 출력을 반환한다.
4. 반환된 출력을 테스트케이스의 출력과 비교한다.
5. 모든 출력 값이 테스트케이스를 통과하면 성공을 알리고, 아니면 실패를 알린다.

실제로 이러한 방식으로 채점이 진행된다고 가정하면 메인 함수는 데이터 가공(테스트케이스 자체가 배열 형태로 메인 함수에 저장되어 있을 가능성도 존재.), 테스트 케이스 검사 등의 작업을 수행한다. 이때 각 언어의 solution 함수는 해당 언어로 작성된 main 함수 내에서 실행될 것이므로 문제마다 언어에 대응되는 초기화 코드뿐만 아니라 main 함수도 작성되어야 할 것으로 보인다.

함수 형식으로 입출력을 진행하는 방식의 장점 및 단점은 다음과 같다.

장점

1. 입력을 가공하는 책임이 문제 작성자 또는 서버에 있으므로 실제 문제 해결 과정에서 사용자의 부담이 줄어들고, 사용자는 문제 해결 자체에 집중할 수 있게 된다.
2. 각 언어에 대한 초기 코드를 제공함으로써 문제 제작자의 의도를 전달할 수 있다. 각 초기 코드에는 입력의 타입 및 출력의 타입이 제시되므로 사용자는 해당 입출력 타입에 기반하여 코드를 작성하게 된다. 이를 통해 사용자의 문제 이해를 도울 수 있을 것으로 보인다.

단점

1. 문제에 대한 언어의 확장성이 낮다. 이전의 가정을 기반으로 채점 시스템이 동작할 때, 각 문제는 해당 문제에서 사용되는 언어들에 대해 초기 코드 및 main 채점 함수를 요구한다. 이때 초기 코드 및 main 함수의 경우 동일한 입력 및 출력 타입을 가진 문제들 사이에서 공유될 수 있기는 하나, 만약 모든 문제에 대해 새로운 언어를 지원해야 한다면 초기 코드 및 main 함수를 자동 생성하고 이를 기존 문제와 매핑하는 복잡한 시스템이 서버 상에 존재하거나, 개발자가 직접 코드를 일일이 작성하여 추가해야 한다. 이는 언어에 대한 컴파일, 실행 과정만 추가하면 되는 BOJ 방식과 다르다.
2. 문제 풀이 내용을 작성하는 함수는 동일한 언어를 기반으로 작성된 메인 함수가 필요하다. main 함수에서는 입력 값을 정의하고 함수를 실행하는 작업을 진행한다. 함수의 결과를 테스트케이스와 비교하여 채점하는 작업이 json 통신을 통해 단일 채점 시스템에서 진행된다고 하더라도 문제 풀이 함수를 실행하기 위해서는 메인 함수에 의해 변수를 전달받아야 하므로 메인 함수가 배제될 수는 없다. 이에 따라 전반적인 시스템 구현이 어려워진다.

2.2 구현 방향 설정

2.1 절에서는 현재 국내에서 운영되는 웹 기반 문제 은행 시스템을 제공하는 서비스 중 BOJ, 프로그래머스의 채점 방식을 예상하여 분석했다.

BOJ 의 표준 입출력 기반 문제 채점 방식은 문제와 프로그래밍 언어가 명확히 구분되어 있어 언어의 확장이 쉽다는 장점이 있었다. 사용자가 표준 입력을 사용 언어의 기능을 통해 입력 받고 현재 실행 결과를 출력하도록 프로그램을 설계함으로써 사용하는 언어와 관계없이 단일 채점 시스템을 기반으로 동작할 수 있다는 점이 매력적이다.

프로그래머스의 방식은 사용자에게 구현할 함수 및 입출력의 구조를 제시하는 것으로, 사용자가 입력을 가공할 필요가 없으므로 문제 해결 자체에 집중할 수 있게 한다. 이에 따라 각 언어별로 초기 코드를 제공하는 것이 인상적이다.

현재 프로젝트에서는 BOJ 의 문자열 기반 표준 입출력을 기반으로 채점 시스템을 구현할 예정이다. 프로그래머스의 방식은 사용자 입장에서 입출력을 구현할 필요가 없어 편리하지만, 서버 내부적으로 사용자가 작성한 함수를 실행하고 채점하는 과정에서 각 언어로 작성된 메인 함수가 추가적으로 요구될 가능성이 높다. 이는 차후 서비스에서 지원하는 언어의 종류를 늘리는데 어려움을 줄 수 있으므로 채택하지 않는다. 대신 각 언어 또는 문제 별로 초기 코드(메인 함수의 역할 수행)를 설정할 수 있도록 구현함으로써 사용자에게 어느 정도 가이드를 줄 수 있도록 설정한다.

2.3 구현 방법 연구

표준 입출력을 이용한 채점 시스템을 구현하기 위해서는 다음과 같은 조건을 만족해야 한다.

1. 유저가 작성한 코드를 서버에서 실행할 수 있어야 한다.
2. 테스트케이스의 input 을 유저의 프로그램에 stdin 으로 전달할 수 있어야 한다.
3. 해당 프로그램에서 출력되는 값을 stdout 이 아니라 채점 시스템에게 전달해야 한다.

위 분석한 요구 조건을 기반으로 각각의 구현 방법을 고려하자.

2.3.1 유저 프로그램 실행

유저의 프로그램은 유저가 온라인에서 작성한 후 채점을 위해 서버로 전송한 코드로, 유저가 선택한 언어에 대응되는 컴파일러 또는 인터프리터에 의해 실행된다. 따라서 서버 또는 채점 시스템은 현재 프로젝트에서 지원하는 언어에 대응되는 컴파일러 및 인터프리터를 보유해야 한다.

서버에 유저의 코드가 도착하는 경우, 유저의 코드는 문자열 상태로 존재한다. 이 코드는 파일에 저장된 후 해당 언어에 대응되는 컴파일러 또는 인터프리터에 의해 실행된다. 이때 해당 실행 과정은 "서버" 프로세스 내부에서 실행되는 것이므로 exec 시스템 콜을 이용할 예정이다.

2.3.2 입력

입력 부분에서는 서버 테스트케이스의 input 을 유저가 작성한 코드에 stdin 으로 전달할 수 있어야 한다. 테스트케이스는 데이터베이스 상에 저장되어 있으며, 서버에서는 문자열 형태로 취급된다. 즉, 문자열 형태의 데이터를 유저가 작성한 코드에 stdin 으로 전달하는 것이 입력 부분의 목적이다.

exec 는 해당 환경의 shell 의 동작을 수행한다. 이때 리눅스의 shell 에는 pipe (|) 라는 기능이 존재한다. pipe 는 프로세스 간 실행 결과를 다른 프로그램에 넘겨줄 때 사용하며, 이전 프로세스의 stdout 을 이후 프로세스의 stdin 과 연결한다. 이러한 특징을 이용하면 다음과 같은 동작을 수행하여 입력을 유저의 프로그램의 stdin 으로 전달하면서 실행할 수 있다.

1. 테스트케이스의 input 을 stdout 으로 출력한다.
2. pipe 를 통해 stdout 을 다음 프로그램의 stdin 으로 변환한다.
3. 유저의 프로그램은 pipe 을 통해 얻은 stdin 을 이용하여 실행된다.

위 과정이 정상적으로 동작하는지 검사하기 위해 다음과 같은 코드를 실행해보자.

```
#include <stdio.h>

int main() {
    int temp;
    int result = 0;
    for(int i = 0; i < 10; i++) {
        scanf("%d", &temp);
        result += temp;
    }

    printf("%d", result);
}

//gcc -o test hello.c
//cat hello.txt | ./test
// 이거 실행한 후 출력하면 됨!
```

(hello.c)

1 2 3 4 5 6 7 8 9 10

(hello.txt, input 역할)

결과는 다음과 같다.

```
blaxsior@DESKTOP-N59DUHU:~/codes/code$ cat hello.txt | ./test
55blaxsior@DESKTOP-N59DUHU:~/codes/code$
```

hello.c 는 10 개의 정수를 scanf 을 통해 입력 받아 전부 더한 후 출력하는 동작을 수행한다. shell 에서는 (1) cat hello.txt 을 통해 10 개의 변수를 stdout 으로 출력하고 (2)이를 pipe 을 이용하여 다음 실행되는 프로그램에게 stdin 으로 전달한다. (3) hello.c 을 컴파일 한 프로그램 test 는 hello.txt 의 내용을

stdin 으로 전달받아 1 ~ 10 까지의 값을 더한 후 결과 55 를 정상적으로 출력하고 있다. 이를 통해 언급된 과정이 정상적으로 동작할 것으로 예상할 수 있다. 실제 input 의 경우 문자열 형식이므로 cat 대신 echo 를 이용하여 파이프로 전달할 수 있도록 구현할 수 있다.

2.3.3 출력

출력 부분 역시 입력과 마찬가지로 프로세스 간 pipe 을 이용하여 값을 전달할 수 있다. 다만 코드가 실행되어 출력하는 마지막 부분이 stdout 이 아니라 서버 영역으로 연결되어야 한다는 점이 다르다. C 언어의 경우 freopen 이나 dup / dup2 을 이용하여 해당 문제를 처리할 수 있다.

다행히도 현재 서버를 구성하는 nodejs 환경에서는 exec⁵의 결과로 stdout 및 stderr 을 기본적으로 얻을 수 있도록 설계되어 있다.

```
const { exec } = require('node:child_process');
exec('cat *.js missing_file | wc -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.error(`stderr: ${stderr}`);
});
```

위 코드는 nodejs 의 “child_process” 설명 문서 중 일부분을 발췌한 것이다. 예제 코드에 따르면 exec 는 실행할 문장(shell 기준) 및 error, stdout, stderr 을 반환하는 콜백 함수를 인자로 받는다. 여기서 현재 프로젝트에 중요한 부분은 exec 의 콜백 함수가 stdout 을 받아올 수 있다는 점이다. exec 가 기본적으로 유저 프로그램이 실행된 결과의 출력 값을 바로 가져올 수 있으므로 출력을 가져오기 위한 방법을 따로 구상하지 않아도 된다.

위 구현 방법을 종합하면 다음과 같다.

1. 유저가 제출한 코드는 서버의 exec 을 이용하여 실행한다.
2. input 은 cat 또는 echo 를 이용하여 stdout 으로 출력하며, 파이프를 통해 유저 코드의 stdin 으로 전달한다.
3. 유저 코드가 실행된 결과인 stdout 은 exec 함수의 콜백을 통해 얻는다.

⁵ https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback

3. 본론

본론 부분에서는 개발에 대한 방향성 및 실제 개발 내역에 대하여 설명한다.

3.1 전체 프로젝트 구조

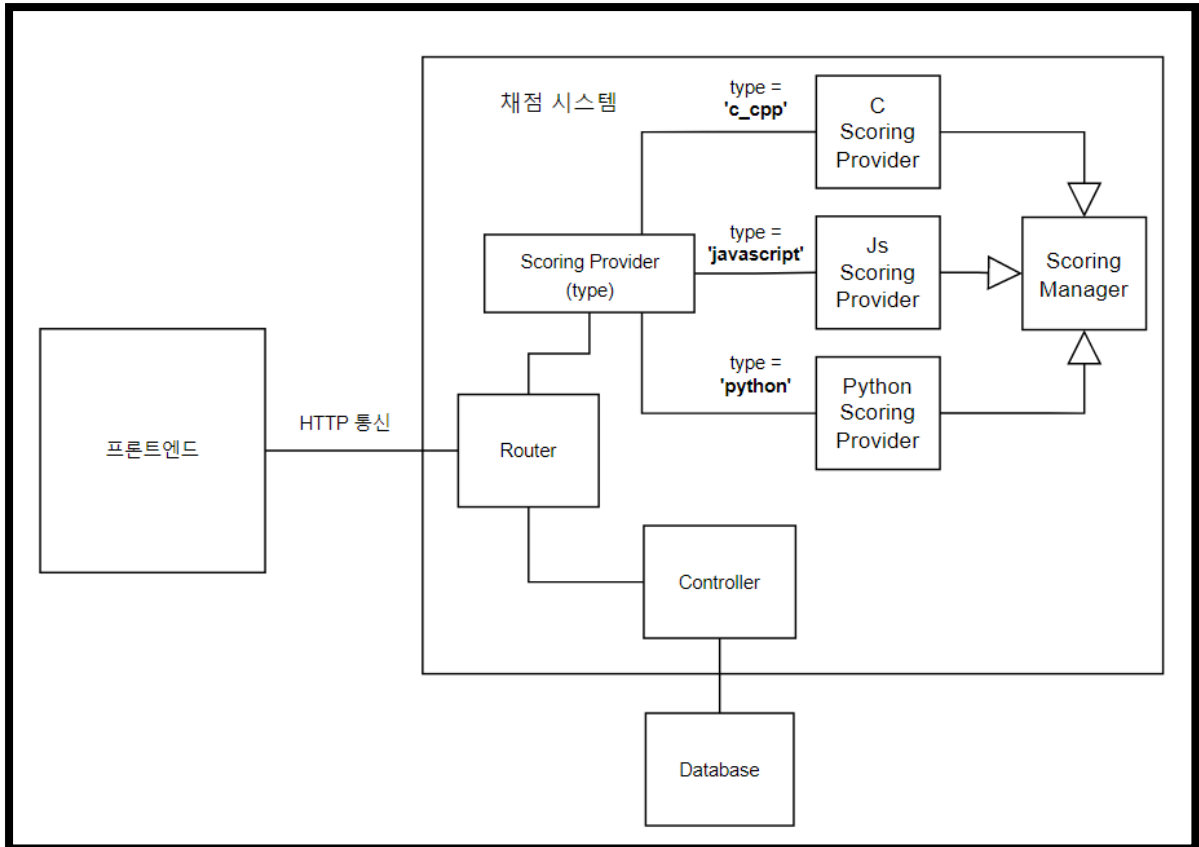


그림 8 프로젝트의 개략적인 구조도

현재 프로젝트에서 구현된 구조는 그림 7의 개략적 구조도와 같다. 사용자는 프론트 엔드에서 문제를 생성하거나 이미 존재하는 문제를 풀 수 있다. 프론트 엔드에서의 각 요청은 서버의 대응되는 경로로 HTTP 통신을 통해 전달되며, 각 경로에서는 데이터베이스의 테이블에 대해 구성된 컨트롤러와 채점시스템인 Scoring Manager을 이용하여 유저가 원하는 데이터를 가공, 이를 반환한다.

현재 프로젝트는 타입스크립트⁶ 기반으로 구현되었으며, 프론트엔드는 react⁷, 백엔드는 express⁸ 기반으로 구현되었으며, 데이터베이스의 경우 sqlite3을 사용하고 있으나 차후 mysql 등 더 전문적인 DBMS로 마이그레이션하기 쉽도록 prisma⁹ ORM을 이용하여 관리한다.

⁶ <https://www.typescriptlang.org/>

⁷ <https://reactjs.org/>

⁸ <https://expressjs.com/>

⁹ <https://www.prisma.io/>

3.2 데이터베이스 설계

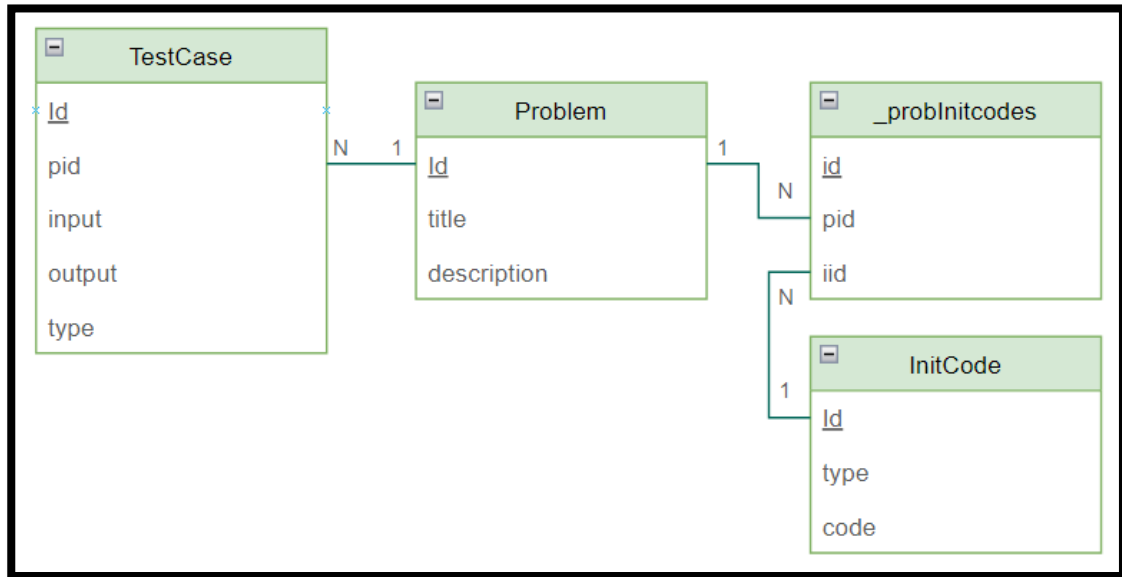


그림 9 데이터베이스 설계

문제, 테스트케이스 및 초기화 코드를 위한 데이터베이스 설계를 진행했다. 각 엔티티에 대한 설명은 다음과 같다.

- Problem: 프로그래밍 문제에 해당한다.
 - Id: 문제의 Id 값.
 - title: 문제의 제목.
 - description: 문제에 대한 설명으로, markdown 형식을 띈다.
- TestCase: 각 문제에 대해 설정한 테스트케이스에 해당한다.
 - id: 테스트케이스를 구분하기 위한 id 값. a
 - pid: 대응되는 문제의 id 값.
 - input: 테스트 케이스의 입력이다. Scoring Manager 을 통해 문제를 채점할 때 입력으로 사용된다.
 - output: 테스트 케이스의 예상된 출력이다. 사용자 코드의 출력과 비교한다.
 - type: 현재 테스트케이스가 맞는 상황에 대한 것인지, 아니면 맞지 않는 상황에 대한 것인지 나타낸다. true 인 경우 유저의 출력 결과와 output 이 동일해야 하나, false 인 경우 유저의 출력 결과가 output 과 다른 값을 가져야 한다.
- InitCode: 사용자가 IDE 을 통해 코드를 구현할 때 초기에 제공하는 입력 값이다.
 - id: 초기 코드의 id 값.
 - type: 현재 코드의 타겟 언어를 의미한다.
 - code: 설정된 초기 코드의 내용을 의미한다.
- _problnitcodes: Problem 과 InitCode 을 M : N 관계로 연결하기 위해 사용된다.

현재 프로젝트에서 데이터베이스는 prisma¹⁰ 라이브러리를 이용하여 관리한다. prisma 는 nodejs 환경에서 데이터베이스를 ORM 방식으로 접근할 수 있도록 도와주는 라이브러리로, 서버에서 사용하는 DBMS 을 변경할 때 DBMS 마다 다른 문법적 차이로 인한 작업을 줄여주고, 기본적인 CRUD 작업을 지원함으로써 데이터베이스를 다루기 편리해지는 장점이 있다.

현재 사용하는 DBMS 는 sqlite3 로, 매우 가볍게 동작하여 테스트 목적 또는 휴대폰 환경에서 간단한 정보를 저장하는데 사용된다. 프로젝트를 구현하는 시점에는 이러한 간단한 DBMS 을 사용할 수 있으나 장기적인 관점에서 보면 mysql, mssql 등 전문적인 DBMS 로 마이그레이션을 진행할 필요가 있다. 이러한 관점에서 차후 마이그레이션 과정에서 발생할 불편을 방지하기 위한 목적으로 현재 프로젝트에서는 prisma 을 채택하고 있다. prisma 를 통해 정의한 스키마는 다음과 같다.

```
model Problem {
  id          Int          @id @default(autoincrement())
  title       String
  description String
  tests       TestCase[]

  initCodes  InitCode[]
}

model TestCase {
  id          Int          @id @default(autoincrement())
  pid         Int
  problem     Problem @relation(fields: [pid], references: [id], onDelete: Cascade)
  input       String?
  output      String
  type        Boolean
  // 테스트케이스가 맞는 경우 혹은 틀리는 경우를 의미
}

model InitCode {
  id          Int          @id @default(autoincrement())
  type        String
  code        String
  problems    Problem[]
}
```

¹⁰ <https://www.prisma.io/>

3.3 백엔드 구현

백엔드 서버는 express 라이브러리를 이용하여 구현된다. 데이터베이스를 연결하기 위한 컨트롤러, 문제 채점을 위한 Scoring Manager 및 사용자의 요청에 대응되는 작업을 수행하는 라우터를 구현하였다.

3.3.1 컨트롤러

3.2 에서 언급한 데이터베이스 테이블 중 Problem, Testcase, Initcode 에 대한 컨트롤러가 구현되어 있다. _probininitcodes 의 경우 단순히 Problem 및 Initcode 을 연결하는데 사용되는데, prisma 라이브러리를 통해 데이터베이스를 설계해두면 자동으로 생성하여 처리해주므로 따로 다루지 않는다.

컨트롤러에서 다루는 코드는 대부분 일반적인 CRUD 작업에 해당하므로 기본적으로 코드 분석을 생략하되, 일부 특이한 점이 있는 경우에만 언급한다. 다음 링크를 통해 코드 내용을 확인할 수 있다.

<https://github.com/blaxsior/codesolveronline/tree/master/backend/src/controller>

ProblemController

problemController 은 Problem 과 관련된 3 개 메서드를 가진 객체로, 메서드는 다음과 같다.

1. getProgramById: 사용자가 특정 문제에 대한 문제풀이 페이지에 방문할 때 해당 문제에 대한 정보를 반환하는데 사용된다.
2. createProblem: 사용자가 프론트 엔드를 통해 생성한 문제를 데이터베이스에 생성할 때 사용된다.
3. getProblemList: 사용자가 문제 목록 페이지에 방문했을 때 존재하는 문제 목록을 제공하기 위해 사용된다.

위 메서드들 중 createProblem 의 코드를 살펴보자.

```
const createProblem = async (prob: IProbInput) => {
  const testcases = prob.testcases.map(it => {
    let type = false;
    if(it.type === 'true')
    {
      type = true;
    }
    return {...it, type};
  });

  const problem = await db.problem.create({
    data: {
      title: prob.title,
      description: prob.description,
      tests: {
        create: testcases
      }
    }
  });
}
```

```

    },
    initCodes: {
      connect: [
        {id: 1},
        {id: 2},
        {id: 3}
      ]
    }
  }
});
return problem !== null;
}

```

createProblem 메서드는 생성 당시에 testcase 정보를 함께 생성한다. data 부분을 잘 살펴보면 tests : { create: testcases } } 코드가 작성된 것을 볼 수 있는데, 이는 problem 을 생성할 때 해당 문제와 관련된 테스트케이스들도 함께 생성한다는 의미이다. 현재 프로젝트에서는 문제와 테스트케이스를 동시에 생성하는 부분까지 구현되며, 테스트케이스를 따로 추가하거나 수정하지는 못한다. 이에 따라 testcasecontroller 에는 테스트케이스를 따로 생성하기 위한 메서드가 아직 정의되어 있지 않다.

TestcaseController

- getTestcaseByPid: 해당 테스트케이스가 적용되는 문제를 id 를 기반으로 식별하여 반환한다.

InitcodeController

- init: 데이터베이스에 미리 준비되어 있는 초기 코드를 삽입할 목적으로 사용된다.
- createInitCodes: 데이터베이스에 초기 코드 목록을 삽입한다.
- deleteInitCodesById: 초기 코드를 id 값을 기반으로 다수 식별하여 제거한다.
- deleteInitCodeById: 초기 코드를 id 값 기반으로 식별하여 제거한다.

현재 InitcodeController 이 구현되어 있기는 하나, 프론트엔드 수준에서 초기 코드를 생성 또는 삭제하는 기능을 구현하고 있지는 않으므로, 서버 내에서 데이터를 삽입하거나 삭제할 목적으로만 사용되고 있다.

3.3.2 채점 시스템

현재 프로젝트는 표준 입출력을 기반으로 동작하며, 채점은 표준 출력에 의한 문자열을 테스트케이스의 출력부와 비교하는 방식으로 수행된다. 따라서 어떤 언어로 코드를 작성하더라도 동일한 채점 방식을 적용할 수 있게 된다. 구체적인 설명은 ScoringManager 클래스를 통해 제시한다.

ScoringManager 클래스

scoringManager 클래스는 채점과 관련된 전반적인 작업을 수행하는 클래스로, 5 개 메서드를 기반으로 모든 언어에 대해 일관된 채점 프로세스를 제공하는 것을 목표로 한다. 파일의 생성, 코드

실행 등의 과정은 **exec** 을 통해 수행된다.

생성자 및 프로퍼티

```
private static readonly temp_dir = resolve('temp'); // C 파일등이 임시로 저장되는 위치
// 차후 반드시 constructor 에서 초기화해야 함!
private readonly id: string; //랜덤 문자열- 파일의 이름
private readonly extension: string; // 현재 언어의 확장자.

private readonly run_str: string; // 코드를 실행할 때 적용할 문자열
private readonly compile_str?: string; // 컴파일할 때 실행할 때 적용할 문자열
private readonly exit_str?: string; // 전체 과정 종료 시 cleanup 에 적용할 문자열

protected constructor({ extension, run, onCompile, onExit }: ISManagerInputParams) {
    this.id = ScoringManager.getRandFileId();
    this.extension = extension;

    this.run_str = run(ScoringManager.temp_dir, this.id);
    this.compile_str = onCompile?.(ScoringManager.temp_dir, this.id);
    this.exit_str = onExit?.(ScoringManager.temp_dir, this.id);
}
```

함수의 생성자에서는 함수 내에서 사용될 다양한 문자열을 초기화한다.

- id: 코드 파일의 이름. 컴파일 또는 인터프리팅을 위해서는 로컬 영역에 파일이 존재해야 하는데, 하나의 서버에서 다수의 사용자를 관리한다는 가정 하에 동일한 이름을 사용하는 경우 사용자 간 중복이 발생할 수 있으므로 getRandFileId 메서드를 이용하여 생성한 랜덤 문자열을 파일 이름으로 할당한다.

- extension: 현재 타겟이 되는 언어의 확장자

- run_str: 생성된 파일을 실행할 때 적용할 문자열

- compile_str: 컴파일이 필요한 언어의 경우 적용할 문자열. 예를 들어 자바스크립트나 파이썬과 달리 C 언어는 파일 이외에 목적 코드를 컴파일해야 한다. 이 부분에서 컴파일 언어와 인터프리터 언어 사이에 차이가 발생하는데, 해당 차이점을 다룰 수 있도록 둔 문자열이다.

- exit_str: compile_str 과 마찬가지로, 컴파일 언어의 경우 컴파일 결과 object 파일이 생성된다. 해당 파일을 삭제할 때 사용되는 문자열이다.

메서드

ScoringManager 이 가진 메서드 목록은 다음과 같다.

- getRandFileId: 파일 이름으로 사용할 랜덤 문자열을 얻는다.
- createCodeFile: 입력된 코드에 대한 파일을 생성한다.
- init: 코드의 생성 및 컴파일 작업 등 코드를 실행하기 전에 필요한 작업을 수행한다.
- run: 코드를 실행하고, 입력된 테스트케이스를 기반으로 결과를 반환한다.
- exit: 생성된 코드 파일, 목적 파일 등을 제거한다.

#01 getRandFileId

```
private static getRandFileId() {  
    return randomBytes(12).toString('hex');  
}
```

파일의 이름으로 사용할 랜덤 문자열을 생성한다. 랜덤 12byte 를 16 진법으로 표기하므로 총 길이는 24 정도 될 것으로 보인다. randomBytes 함수의 경우 nodejs 의 "crypto" 모듈에서 기본적으로 지원한다.

#02 createCodeFile

```
private async createCodeFile(code: string, timeout: number = 3000):  
Promise<boolean> {  
    let success = true;  
    try {  
        await exec(`cat >  
${ScoringManager.temp_dir}/${this.id}.${this.extension} <<  
EOF\n${code}\nEOF`, { timeout });  
    }  
    catch (err) {  
        console.error("create code file");  
        success = false;  
    }  
    return success;  
}
```

createCodeFile 메서드에서는 입력된 코드를 파일로 만드는 작업을 수행한다. code 파라미터를 통해 전달된 코드는 **[cat > 파일_이름 << EOF ~ EOF]** 명령에 삽입되어 파일 형태로 생성된다. 이 커맨드 명령은 cat 을 통해 파일_이름에 해당하는 파일에 내용을 쓰되, EOF 가 나타날 때까지 계속 작성한다는 의미로 해석되며, 단순히 입력된 코드를 담고 있는 파일을 생성하는 작업을 수행하는 것이므로 nodejs 의 파일 시스템을 이용하는 방법도 유효할 것으로 보인다. 현재 프로젝트에서는 init, run, exit 등에서의 일관성을 위해 exec 함수를 이용하여 처리했다.

#03 init

```
async init(code: string, timeout: number = 3000): Promise<boolean> {  
    let success = await this.createCodeFile(code); // 파일 생성 여부 검사  
    if (success && this.compile_str) {  
        // 컴파일이 필요한 경우에는 컴파일을 진행.  
  
        try {  
            await exec(this.compile_str, { timeout });  
        }  
        catch (err) {  
            console.error("init");  
            console.log(err);  
            success = false;  
        }  
    }  
}
```

```

    }
  }
  return success;
}

```

코드를 파일로 생성하고 compile_str 이 정의되어 있는 경우 해당 명령을 exec 를 통해 실행한다. C 언어와 같은 컴파일 언어의 경우 생성자에 onCompile 메서드를 넘겨줌으로써 compile_str 을 생성할 수 있으며, 이를 이용하여 컴파일을 진행한다.

#04 run

```

async run(tc: TestCase, timeout: number = 5000) {
  let success = true;
  try {
    const { stdout, stderr } = await exec(`${tc.input ? `echo "${tc.input}" |` : ""} ${this.run_str}`, { timeout });
    if ((stdout == tc.output) !== tc.type) {
      success = false;
    }
  }
  catch (err) {
    // 나중에 에러 이유 등도 알 수 있도록 할 수 있다면...
    console.error(err)
    success = false;
  }
  return success;
}

```

컴파일 된 파일을 실행하고, stdout 형식으로 반환된 문자열을 테스트케이스의 output 과 비교하여 테스트케이스의 타입과 일치하는지 정보를 Boolean 타입으로 반환한다. 현재 코드에서 중요한 부분은 exec 의 내부에서 실행되는 명령이다.

```
`${tc.input ? `echo "${tc.input}" |` : ""} ${this.run_str}`, { timeout }
```

위 코드는 exec 에 의해 실행되는 커맨드라인 명령에 해당한다. 프로그래밍 문제 중에는 입력 값이 존재하지 않는 경우가 있는데, 이 경우를 대비하여 3 항 연산자를 이용해서 echo 실행 여부를 결정한다. echo 는 자신이 입력한 문자열을 그대로 출력해주는 커맨드라인 명령이다. run 메서드에서는 테스트케이스의 입력 값을 echo 을 통해 출력하되, 파이프를 이용하여 this.run_str 의 입력으로 사용하도록 구현한다. 앞에서 run_str 은 코드를 실행할 때 사용되는 문자열이라고 언급했다. 예를 들어 파이썬 3 버전의 코드를 실행하기 위해서는 python3 hello.py 을 커맨드 라인에 입력할 것이다. 따라서 echo "hello" | python3 hello.py 명령은 python3 로 실행된 코드에 대해 input 으로 hello 문자열을 전달하는 코드가 된다.

#05 exit

```

async exit() {
  let success = true;

```

```

        try {
            const { stdout, stderr } = await exec(`rm
${ScoringManager.temp_dir}/${this.id}.${this.extension}; ${this.exit_str} ??
""`);
        }
        catch (e) {
            console.error("exit");
            success = false;
        }
        return success;
    }
}

```

exit 메서드는 생성된 파일을 제거하기 위한 메서드로, 추가적인 파일이 생성되는 경우 exit_str 에 해당 파일을 제거하기 위한 커맨드라인 명령을 제공할 수 있다.

ScoringManager 의 생성자는 protected 로 정의되어 있어 인스턴스를 생성할 수 없으므로, 특정 언어에 대한 채점 시스템을 상속을 통해 구현하도록 강제한다. 상속한 클래스에서는 extension, run, onCompile, onExit 에 대응되는 함수를 정의하여 ScoringManager 의 생성자에 넘겨주는 방식으로 채점 시스템을 사용할 수 있게 된다. 예시는 다음과 같다.

```

import { ISManagerInputParams, ScoringManager } from "../ScoringManager.js";

export class CScoringManager extends ScoringManager {
    constructor() {
        const params: ISManagerInputParams = {
            extension: 'c',
            run: (path, id) => `${path}/${id}`,
            onCompile: (path, id) => `gcc -o ${path}/${id} ${path}/${id}.c`,
            onExit: (path, id) => `rm ${path}/${id}`
        };

        super({ ...params })
    }
}

```

ScoringProvider 함수

```

export const ScoringProvider = (lang: Lang|string): ScoringManager|null => {
    switch(lang)
    {
        case "c_cpp":
            return new CScoringManager();
        case "javascript":
            return new JsScoringManager();
        case "python":
            return new PythonScoringManager();
        default:
            return null;
    }
}

```

```
}  
}
```

ScoringProvider 은 ScoringManager 에 대한 팩토리 메서드로, 언어 이름을 인자로 받아 대응되는 언어에 대한 ScoringManager 을 반환하도록 구현된다. 현재 프로젝트에서는 gcc 기반의 c / c++, nodejs 기반의 자바스크립트 및 python3 기반의 파이썬 언어를 지원하고 있다.

3.3.3 라우터

제공하는 경로는 다음과 같다.

- -[GET] /p/problem/:id: 특정 문제 및 문제에 대한 초기 코드를 제공한다.
- -[GET] /p/list/:pno: 문제 목록을 반환한다. pno 는 페이지 번호를 의미한다.
- -[POST] /p/create: 제공된 데이터를 기반으로 문제를 생성한다.
- -[POST] /p/score: 제공된 코드를 기반으로 채점을 수행한다.

위 경로 중 /p/score 에 대응되는 메서드만 잠시 살펴보자.

#scoreCode

```
const scoreCode: RequestHandler = async (req, res, next) => {  
  const data = req.body as IPSInput;  
  const codeManager = ScoringProvider(data.type);  
  // ... 여러가지 에러 처리 코드들  
  const success = await codeManager.init(data.code);  
  // 에러 처리 코드...  
  const testcases = await TestcaseController.getTestcasesByPid(id);  
  // ... 에러처리 코드  
  const re_arr: boolean[] = [];  
  
  for (let tc of testcases) {  
    const result = await codeManager.run(tc);  
    re_arr.push(result);  
  }  
  const count = re_arr.filter(it => it === true).length;  
  if (count === re_arr.length) {  
    await codeManager.exit();  
    return res.send({ message: "성공입니다!" });  
  }  
  else {  
    await codeManager.exit();  
    return res.send({ message: "실패입니다!" });  
  }  
}
```

scoreCode 메서드는 /p/score 경로에 대응되는 메서드이다. 사용자가 입력한 데이터를 기반으로 대응되는 언어의 ScoringManager 과 대응되는 문제의 테스트케이스 목록을 얻는다. 이후 테스트케이스의 개수만큼 run 을 반복한 후, 결과를 filter 을 통해 모든 테스트케이스가 맞았는지 검사한다. 맞는 경우 "성공입니다" 메시지를, 틀린 경우 "실패입니다" 메시지를 반환한다.

3.4 프론트엔드 구현

프론트엔드는 현재 총 4 개 페이지로 구현되어 있다. 각 경로는 다음과 같다.

- / : 메인 페이지
- /list : 문제 목록 페이지
- /create : 문제 생성 페이지
- /solve/:id : 문제 풀이 페이지

현재 프론트엔드는 react 라이브러리를 기반으로 하고 있으며, react-router 라이브러리를 이용하여 client side routing 을 수행하고 있다. 추가적으로 현재 프론트엔드의 레이아웃 및 일부 css 설정은 시간적 문제로 인해 동일 학기 "소프트웨어 공학 개론" 수업에서 본인이 만든 컴포넌트¹¹를 사용하고 있음을 알린다.

프론트엔드에서 발생하는 HTTP 요청은 rewrite 규칙에 의해 백엔드로 전달된다.

```
app.use('/api',
  createProxyMiddleware({
    target: 'http://localhost:5000',
    changeOrigin: true,
    pathRewrite: { '^/api': '/' }
  })
```

(setupProxy.js 파일에 정의되어 있다.)

백엔드는 /api 로 시작되는 주소를 통해 접근할 수 있으며, 해당 주소는 rewrite 규칙에 의해 서버의 '/' 경로를 요청하게 된다. 예를 들어 /api/hello 라는 주소는 rewrite 규칙에 의해 /api 가 제거되고 /hello 라는 경로로 서버에 요청하게 된다.

구체적인 페이지 구현 사항은 4. 구현 결과 파트를 참고하라.

¹¹ <https://github.com/fish9903/Bus-Tour/tree/main/client/client2/src/layout>

4. 구현 결과

구현 결과에서는 사용자가 사용하는 프론트엔드의 각 페이지를 기준으로 페이지가 동작하는 모습을 제시한다. 페이지는 메인 페이지, 문제 목록 페이지, 문제 생성 페이지, 문제 풀이 페이지가 존재한다.

4.1 메인 페이지

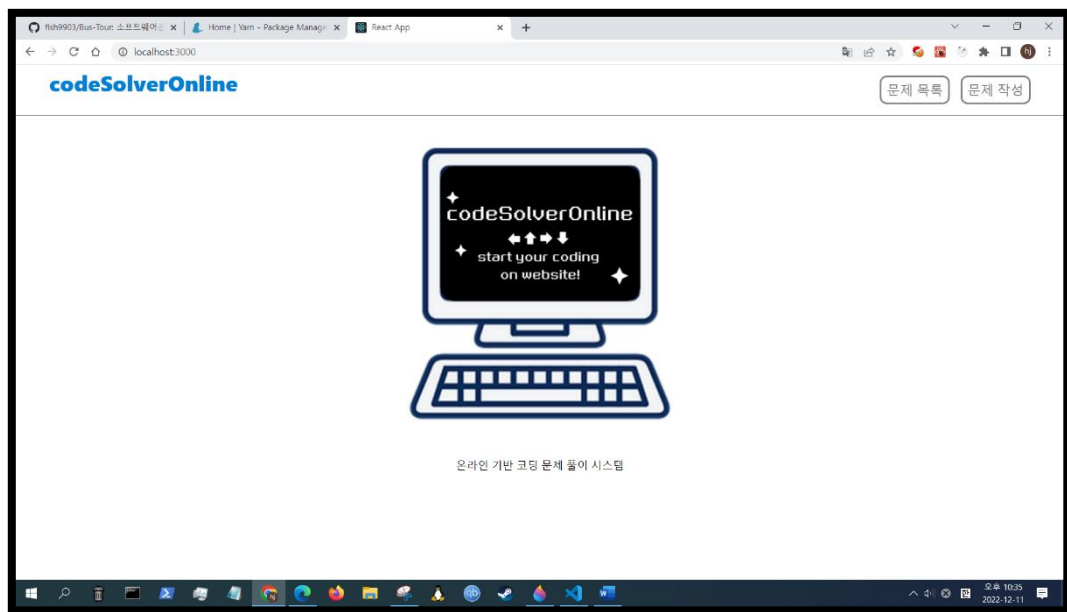


그림 10 메인 페이지

메인 페이지는 현재 별다른 기능이 없이 본인이 canva¹² 사이트를 이용하여 디자인한 컴퓨터 모양의 로고¹³를 보여주고 있다. 차후 시스템이 확장되는 경우 로그인 또는 사용자 별 추천 문제 등을 표시하는 것도 좋을 것 같다.

¹² <https://www.canva.com/>

¹³컴퓨터 모양 로고: <https://url.kr/oxy4is>

4.2 문제 목록 페이지

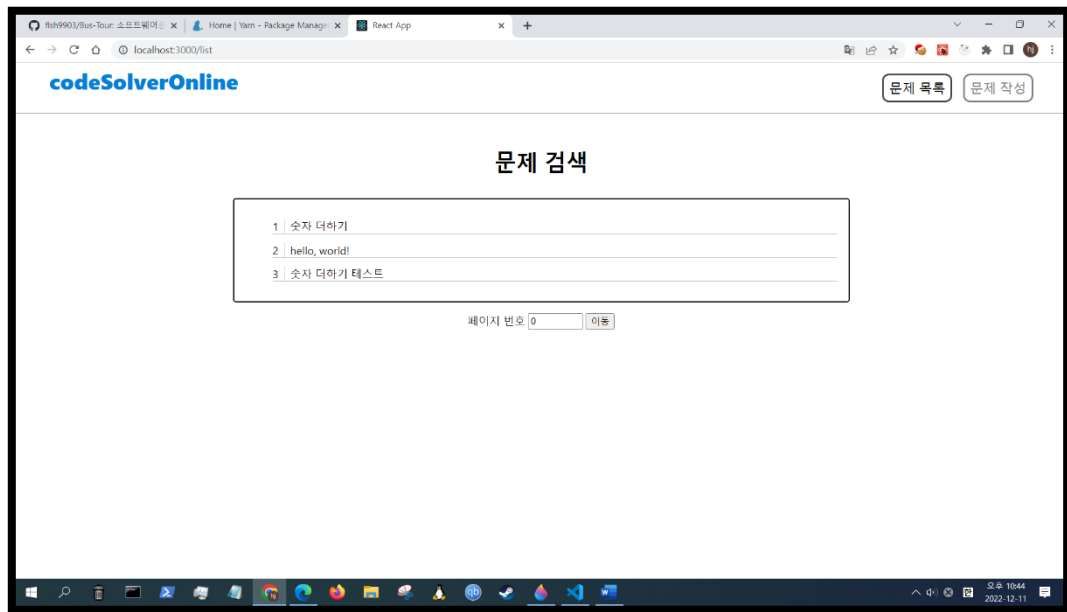


그림 11 문제 목록 페이지

문제 목록 페이지에서는 현재 데이터베이스 상에 존재하는 문제 목록을 가져와 리스트를 출력한다. 리스트는 한 번에 10 개까지만 검색할 수 있으며 페이지 번호를 변경하여 다른 문제 목록을 볼 수 있다.

```
try {
  const result = await axios.get(`/api/p/list/${id}`, {
    timeout: 3000 // 3 초 안에 처리 안되면 에러로 처리됨
  });
  if (result.status !== 200) {
    return {pno,data: []};
  }
  else {
    return {pno,data:result.data};
  }
}
```

list 페이지에 방문하면 대응되는 loader 이 실행된다. loader 함수에서는 /api/p/list/:id 경로로 GET 요청을 보낸다. 해당 요청은 백엔드의 동일한 경로에 대응되며, 존재하는 문제 리스트를 받는다.

사용자는 문제 목록 중 특정 문제를 클릭하여 해당 문제에 대한 문제 풀이 페이지로 이동할 수 있다.

4.3 문제 생성 페이지

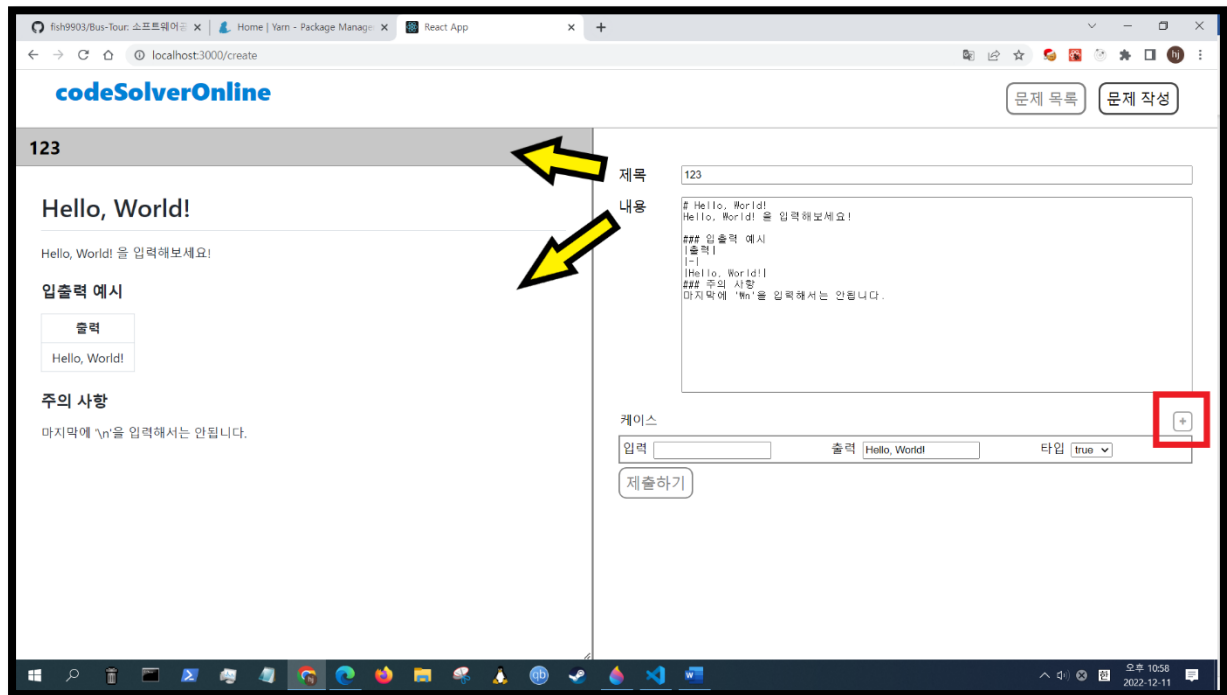


그림 12 문제 생성 페이지

문제 생성 페이지는 크게 2 개의 부분으로 나뉜다.

1. DescSection: 좌측 부분으로, DescEditSection 에서 작성한 마크다운 형식을 파싱하여 보여준다. 이때 파싱된 내용은 'remark-gfm'¹⁴ 플러그인을 적용하여 github 스타일의 마크다운 형식을 띤다. 마크다운 형식의 텍스트는 react-markdown¹⁵ 라이브러리에 의해 파싱된다.
2. DescEditSection: 우측 부분으로, 제목, 내용 및 문제에 대한 테스트케이스를 작성할 수 있다. 내용은 마크다운 형식으로 작성해야 하며, 작성한 내용은 좌측의 DescSection 을 통해 확인할 수 있다. 테스트케이스의 경우 최소 하나 이상 작성해야 하며 입력, 출력 및 테스트케이스의 타입을 정의할 수 있다. + 버튼을 눌러 테스트케이스를 추가할 수 있다.

¹⁴ <https://www.npmjs.com/package/remark-gfm>

¹⁵ <https://www.npmjs.com/package/react-markdown>

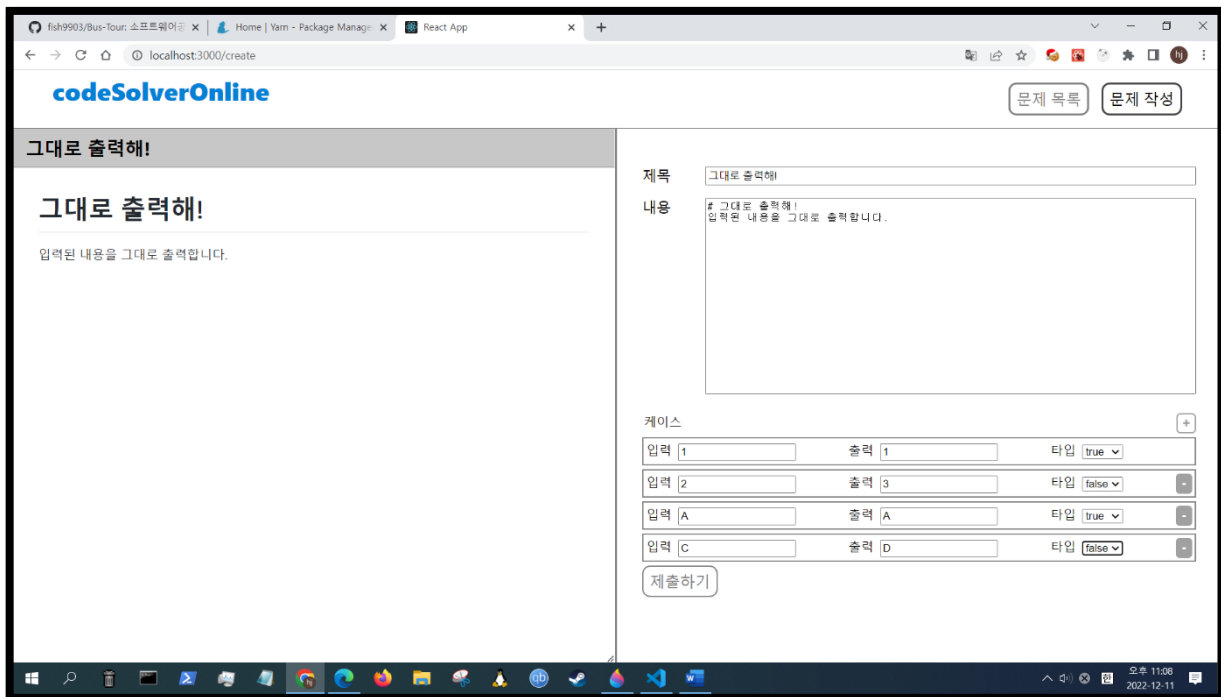


그림 13 문제 생성 페이지에서 테스트 케이스를 추가한 모습

그림 13 은 문제를 생성할 때 테스트케이스를 추가하는 상황을 보여준다. + 버튼을 클릭하여 추가한 테스트케이스들은 - 버튼을 가진 채로 추가되어 개별적으로 제거할 수 있도록 구현되어 있다. 이때 최상위 테스트케이스는 - 버튼이 존재하지 않는데, 이는 최소 테스트케이스를 하나 만들도록 강제하고자 하는 목적이 있다.

```
export const action: ActionFunction = async ({ request, params }) => {
  const data = (await request.formData()).entries();
  const darr = [...data] as string[][];
  const urlparams = new URLSearchParams(darr).toString();
  console.log(urlparams);
  try {
    const result = await axios.post('/api/p/create', urlparams, {
      headers: {
        "Content-Type": "application/x-www-form-urlencoded",
      }
    });
  }
});
```

제출하기 버튼을 클릭하여 form 을 제출하는 경우 페이지에 대응되는 action 함수가 호출된다. action 함수에서는 제출한 form 으로부터 FormData 클래스를 추출하여 entries 형식으로 나열한다. 타입스크립트를 사용하는 경우 entry 에 의한 결과는 UrlSearchParams 에 의해 바로 파싱될 수 없는데, 이는 formData 에 문자열뿐만 아니라 raw 파일 정보도 포함될 수 있기 때문이다. 현재 프로젝트에서는 파일 데이터를 form 에서 다루지 않으므로 문자열 배열로 형변환하여 UrlSearchParams 에 전달한 후 문자열로 반환 받는다. 위와 같은 동작은 express 서버에서 Content-Type: "multipart/form-data"인 데이터를 바로 파싱하지 못하기 때문에 수행한다. 참고로 "multipart/form-data" 타입에 대한 파싱을

지원하는 `multer`¹⁶이라는 라이브러리가 존재하지만, 현재 시스템에서는 프론트엔드 부분에서 아직 파일을 다루지 않으므로 사용하지 않았다.

4.4 문제풀이 페이지

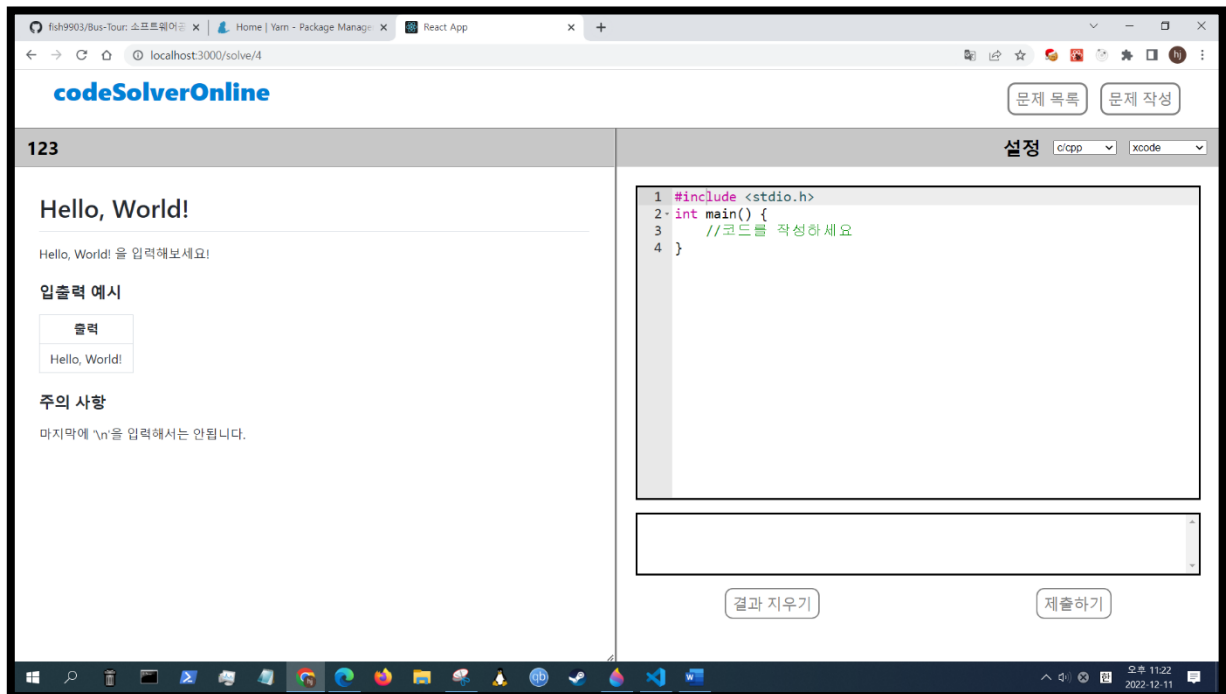


그림 13 문제풀이 페이지

그림 13은 문제풀이 페이지를 보여준다. 문제풀이 페이지는 2개의 섹션으로 구성되어 있다.

1. DescSection: 문제에 대한 설명이 서술되어 있는 섹션이다. 문제 생성 페이지에서 사용된 컴포넌트와 동일하며, 문제 목록 페이지에서 클릭한 대응되는 문제에 대한 설명을 보여준다.
2. CodeEditSection: 코드를 작성하는 섹션이다. 백엔드 서버에서 `initcodeController`를 통해 데이터베이스 상에 삽입한 초기 코드가 코드 작성 부분에 나타난다. 사용자는 "설정" 부분에서 사용할 언어 및 스킨을 결정할 수 있다. '제출하기' 버튼을 클릭하는 경우 백엔드 서버에 문제 채점을 요청하게 되며, 채점 결과는 코드 작성 란 아래의 박스 부분에 나타난다.

¹⁶ <https://www.npmjs.com/package/multer>



그림 14 성공한 경우

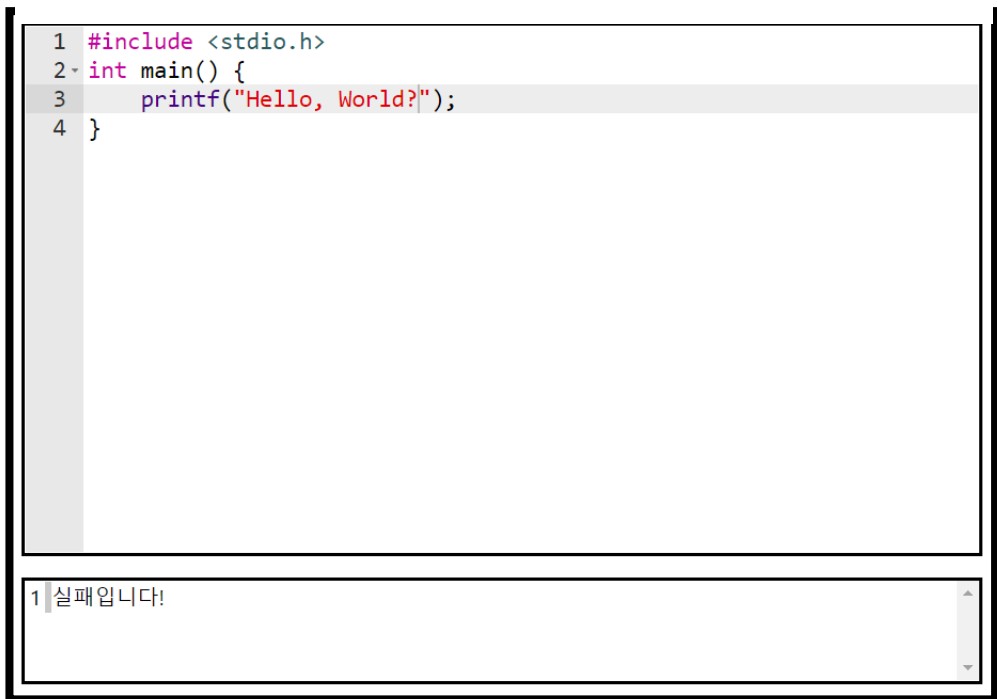


그림 15 실패한 경우

사용자는 코드 섹션의 IDE 에 코드를 작성하여 제출버튼을 누름으로써 자신의 코드를 채점할 수 있다. 이때 모든 테스트케이스를 통과하는 경우 "성공합니다" 메시지를, 하나라도 실패하는 경우 "실패입니다" 메시지가 아래 창에 나타난다.

코드 섹션 위 설정에서는 사용하는 언어와 적용되는 스킨을 변경할 수 있다.

현재 지원하는 목록은 다음과 같다. 프론트엔드의 입력부는 현재 react-ace¹⁷ 라이브러리를 이용하여 구현된다. 해당 라이브러리는 ace editor¹⁸ 을 리액트의 컴포넌트로 구현한 것이다. ace editor 에서 지원 언어 또는 테마를 적용하기 위해서는 관련된 파일을 import 해야 하며, 현재는 아래 명시되어 있는 설정에 대한 파일들만 import 되어 있는 상태다.

- 언어: c/cpp, javascript, python
- 테마: xcode, cloud9_night



그림 16 언어: 파이썬, 스킨: cloud9_night 로 변경한 모습

위 언급된 동작들을 통해 현재 구현한 시스템이 올바르게 동작하고 있음을 알 수 있다.

¹⁷ <https://www.npmjs.com/package/react-ace>

¹⁸ <https://ace.c9.io/>

4.5. 실행 방법

현재 프로젝트를 로컬에서 실행하기 위해서는 최소한 nodejs, yarn¹⁹, gcc, python3 가 설치되어 있어야 하며, 환경 차이에 따라 ScoringManager 에 제공되는 인자들의 값을 변경해야 할 수 있다.

```
# 파일 설치
git clone https://github.com/blaxsior/codesolveronline # 현재 프로젝트의 파일들을 클론
cd codesolveronline # 프로젝트 파일로 이동
# 프론트엔드 설치
cd frontend # 프론트엔드 폴더로 이동
yarn install # 프론트엔드 의존성 설치
cd ..
# 백엔드 설치
cd backend # 백엔드 폴더로 이동
yarn install # 백엔드 의존성 설치
yarn prisma generate # prisma
# .dev 파일 생성
cat> .env << EOF
DATABASE_URL="file:./dev.db"
EOF
cd ..
# 프론트엔드 실행
cd frontend && yarn start && cd ..
# 백엔드 실행
cd backend && yarn run dev && cd ..
```

¹⁹ <https://yarnpkg.com/>

5. 결론 및 향후 연구

현재 프로젝트에서는 웹 기반 IDE 문제은행 서비스를 운영하고 있는 BOJ 및 프로그래머스의 채점 방식을 추정 및 분석하고, 분석 결과를 기반으로 간단하게 데이터베이스 설계, 테스트케이스 입력, 문자열 기반 채점 시스템 구현 및 프론트엔드 연동을 통한 실행 과정을 수행했으며, 추측한 방식이 성공적으로 동작함을 확인할 수 있었다.

향후 연구를 추가적으로 진행하는 경우 기존에 진행했던 "CS 웹 IDE 기반 문제은행 연구"에서 다뤘던 tree-sitter 라이브러리를 이용하여 현재 구현한 시스템 상에 lint 기능을 추가하거나, 함수 기반 입출력 방식에서 각 언어에 대응되는 함수가 추가적으로 요구되어 확장성이 감소하는 문제를 극복할 다른 방안 등을 고려할 수 있을 것 같다. 추가적으로 서버에 피해를 입힐 수 있는 악의적인 코드 작성을 방지하기 위해 docker 을 통해 채점 서버를 격리하는 방법도 추가 진행 사항으로 알맞아 보인다.

현재 프로젝트에서 작성된 모든 코드는 아래 제시된 본인의 github 페이지에서 볼 수 있다.

<https://github.com/blaxsior/codesolveronline>