W03
IDE와 코딩 에이전트

바이브 코딩 그라운드

바이브 코딩의 발전

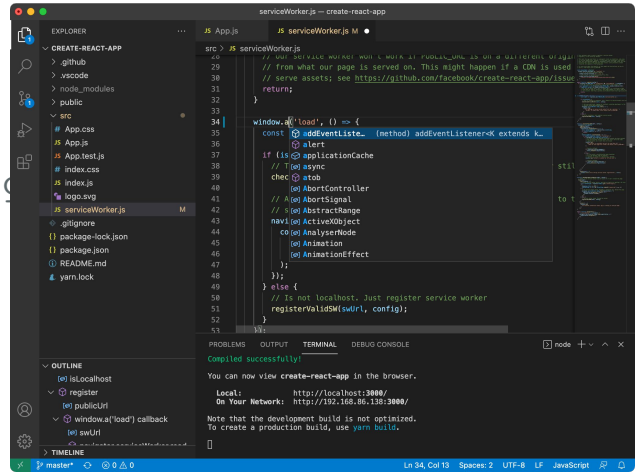
1. LLM 지시 -> 인간 수행
    a. AI에게 만들고 싶은 프로그램을 말하면 개발하는 방법을 알려준다
    b. AI가 지시하는 대로 모든 개발을 수행하는 단계를 하나씩 인간이 직접 진행한다.
    c. 오류 메세지가 발생하면 그것 역시 LLM에게 물어 시키는 대로 개편한다
2. MCP의 등장 -> 파일 시스템 연결
    a. Desktop 어플리케이션이 파일 시스템에 연결되어 파일을 직접 생성 수정한다.
    b. 해당 코드를 인간이 직접 실행하고 검토하면서 문제들을 다시 AI에게 전달한다.
    c. 직접 프로젝트 파일들을 보며 검토 및 수정이 가능하다

바이브 코딩의 발전

3. Coding Agent의 등장 -> 터미널에서 직접 대화

- Coding 전문 Agent에 접속하여 명령을 내리면 코딩에 특화된 절차를 수행한다.
- 내린 명령에 필요한 코드 작성, 수정, 실행, 검증 및 오류 수정을 자동으로 수행한다.
- 터미널이라고 하는 환경에서 사용자가 명령을 입력하면 개발 환경에서 직접 코드를 변경
- Claude Code, GPT Codex, Gemini CLI 등

IDE(통합 개발 환경) 이란?



IDE = 코드를 작성하기 위한 '작업실'

- 코드 편집기: 코드를 작성하는 공간 (워드프로세서 같은
- 파일 탐색기: 프로젝트의 모든 파일을 한눈에 보고 관리
- 터미널: 컴퓨터에게 직접 명령을 내리는 창
- 디버거: 코드에서 문제가 생긴 위치를 찾아주는 도구

왜 IDE가 필요한가?

- 메모장에서도 코드를 작성할 수 있다
- 하지만 오타를 찾기 어렵고, 파일을 일일이 열어야 하고, 실행하려면 다른 프로그램이 필요하다
- IDE는 이 모든 것을 한 곳에서 할 수 있게 해준다

"IDE는 요리사의 주방이다 - 칼, 도마, 불, 재료가 모두 손닿는 곳에 있어야 요리가 된다"

우리가 하는 방식은? 4. IDE 통합 에이전트

- IDE안에 에이전트가 직접 연결되어 사용자의 명령을 수행
- 코드 뿐 아니라 IDE의 다양한 개발 관련 기능과 연동되어 다양한 일들을 수행 가능
- VSCode Extention, Cursor, Antigravity 등

이전 방식과의 차이

- 기존: 창을 왔다갔다 해야 했다 (AI 대화창 ↔ 코드 편집기 ↔ 터미널)
- 이 방식: 하나의 화면에서 모든 것이 연결된다

"AI가 내 작업 환경 안에 들어와 앉았다"

커서(Cursor)란?

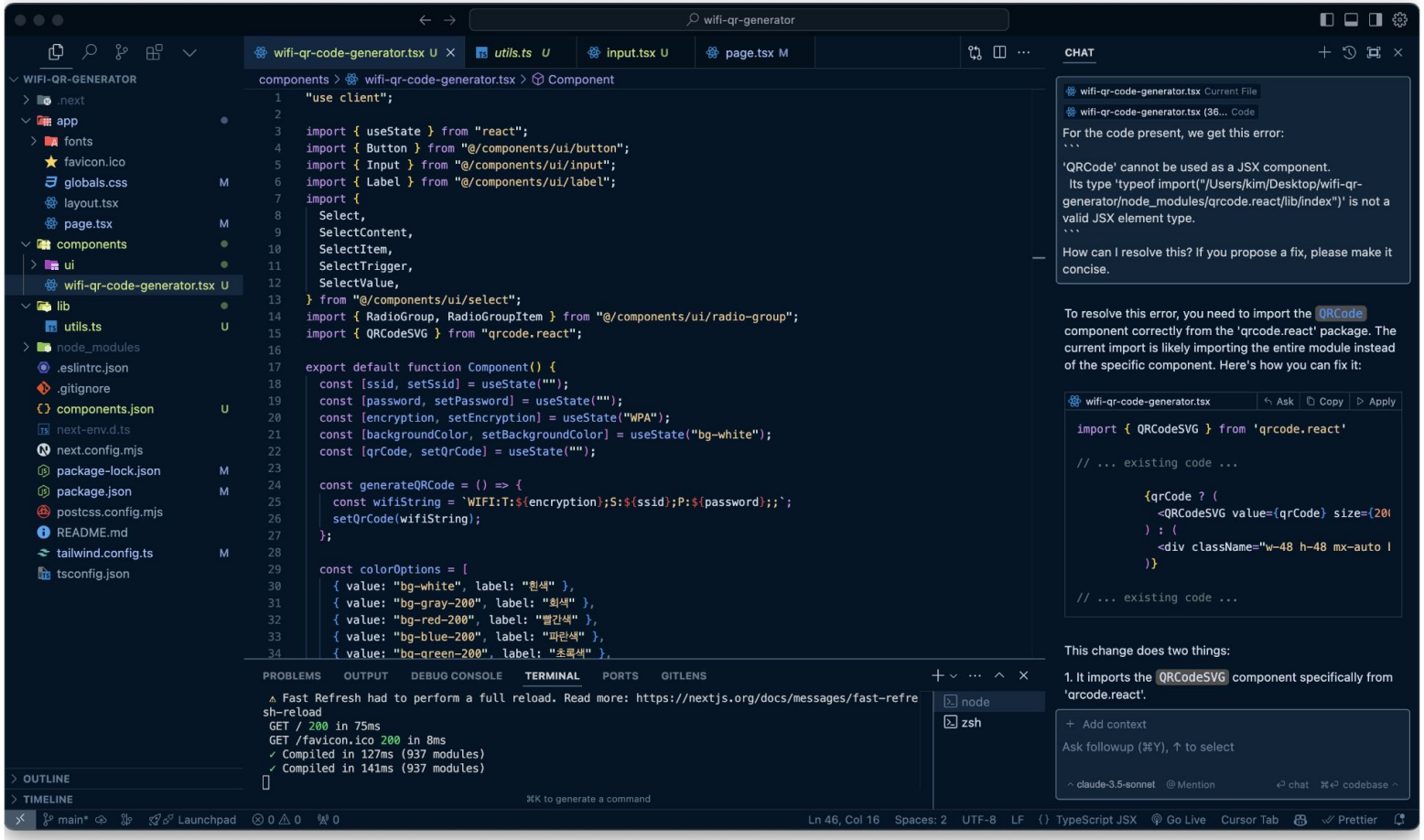
Cursor = VS Code + AI 에이전트가 결합된 IDE

- VS Code: 전 세계에서 가장 많이 쓰이는 무료 코드 편집기
- Cursor는 VS Code를 기반으로 만들어져서 기존 사용법을 그대로 활용 가능
- 여기에 AI 에이전트가 내장되어 대화하면서 코드를 작성할 수 있다

Cursor가 성공한 이유

- VS Code의 익숙함을 유지하면서
- AI 기능이 '플러그인'이 아닌 '핵심 기능'으로 통합되어 있다
- 코드를 보면서 바로 AI에게 질문하고, AI가 바로 코드를 수정한다

"기존 도구에 AI를 붙인 것이 아니라, AI를 위해 처음부터 설계된 IDE"


wifi-qr-generator
WIFI-QR-GENERATOR
.next
app
fonts
favicon.ico
globals.css M
layout.tsx
page.tsx M
components
ui
wifi-qr-code-generator.tsx U
lib
utils.ts U
node_modules
.eslintrc.json
.gitignore
components.json U
next-env.d.ts
next.config.mjs
package-lock.json M
package.json M
postcss.config.mjs
README.md
tailwind.config.ts M
tsconfig.json
OUTLINE
TIMELINE
wifi-qr-code-generator.tsx U
utils.ts U
input.tsx U
page.tsx M
components > wifi-qr-code-generator.tsx > Component
"use client";
import { useState } from "react";
import { Button } from "@/components/ui/button";
import { Input } from "@/components/ui/input";
import { Label } from "@/components/ui/label";
import {
Select,
SelectContent,
SelectItem,
SelectTrigger,
SelectValue,
} from "@/components/ui/select";
import { RadioGroup, RadioGroupItem } from "@/components/ui/radio-group";
import { QRCodeSVG } from "qrcode.react";
export default function Component() {
const [ssid, setSsid] = useState("");
const [password, setPassword] = useState("");
const [encryption, setEncryption] = useState("WPA");
const [backgroundColor, setBackgroundColor] = useState("bg-white");
const [qrCode, setQrCode] = useState("");
const generateQRCode = () => {
const wifiString = `WIFI:T:${encryption};S:${ssid};P:${password};;`;
setQrCode(wifiString);
};
const colorOptions = [
{ value: "bg-white", label: "흰색" },
{ value: "bg-gray-200", label: "회색" },
{ value: "bg-red-200", label: "빨간색" },
{ value: "bg-blue-200", label: "파란색" },
{ value: "bg-green-200", label: "초록색" },
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
GITLENS
⚠ Fast Refresh had to perform a full reload. Read more: https://nextjs.org/docs/messages/fast-refresh-reload
GET / 200 in 75ms
GET /favicon.ico 200 in 8ms
✓ Compiled in 127ms (937 modules)
✓ Compiled in 141ms (937 modules)
node
zsh
⌘K to generate a command
CHAT
wifi-qr-code-generator.tsx Current File
wifi-qr-code-generator.tsx (36... Code
For the code present, we get this error:
'QRCode' cannot be used as a JSX component.
Its type 'typeof import("/Users/kim/Desktop/wifi-qr-generator/node_modules/qrcode.react/lib/index")' is not a valid JSX element type.
How can I resolve this? If you propose a fix, please make it concise.
To resolve this error, you need to import the QRCode component correctly from the 'qrcode.react' package. The current import is likely importing the entire module instead of the specific component. Here's how you can fix it:
wifi-qr-code-generator.tsx
Ask
Copy
Apply
import { QRCodeSVG } from 'qrcode.react'
// ... existing code ...
{qrCode ? (
<QRCodeSVG value={qrCode} size={20(
) : (
<div className="w-48 h-48 mx-auto l
)}
// ... existing code ...
This change does two things:
1. It imports the QRCodeSVG component specifically from 'qrcode.react'.
Add context
Ask followup (⌘Y), ↑ to select
claude-3.5-sonnet
@ Mention
chat
codebase
main*
Launchpad
Ln 46, Col 16
Spaces: 2
UTF-8
LF
TypeScript JSX
Go Live
Cursor Tab
Prettier

커서의 구조

1. 에디터 (왼쪽/중앙)

- 실제 코드를 작성하고 편집하는 공간
- AI가 제안한 코드가 여기에 직접 반영된다

2. 채팅 패널 (오른쪽)

- AI 에이전트와 대화하는 공간
- "이 함수 리팩토링 해줘", "에러 원인 분석해줘" 등 자연어로 명령

3. 컨텍스트 연결

- 채팅에서 @파일명, @폴더명으로 특정 파일을 지정 가능
- AI가 해당 파일의 내용을 직접 읽고 수정안을 제시

"AI가 내 프로젝트 전체를 보면서 대화할 수 있다"

개발을 위해서 코딩 에이전트는

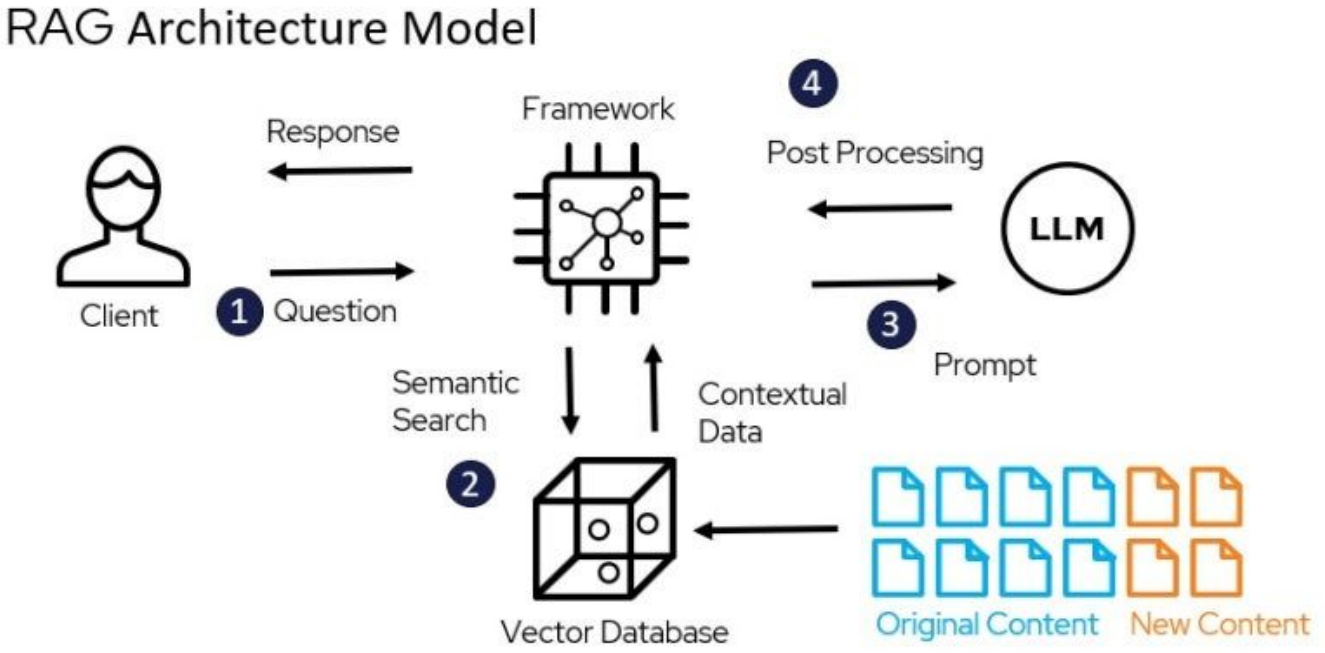
무엇을 알아야 하는가?

1. 목적: 어떤 행위를 해야하는가? (프롬프트 명령)
2. 실제 코드: 이 프로젝트는 어떤 구조로 어떤 코드가 작성되어 있는가 (파일 내용)
3. 히스토리: 지금까지 어떤 일들이 일어났으며 어떤 의사결정을 했는가? (과거 대화내용)

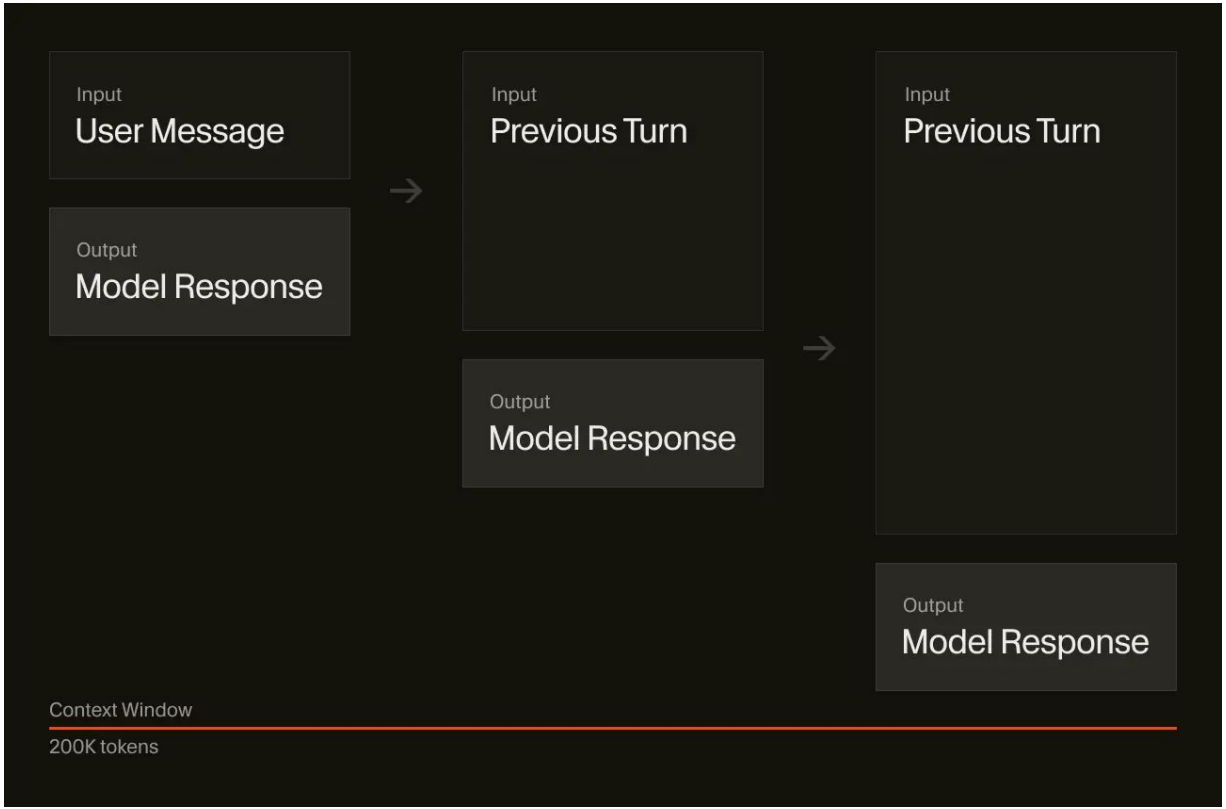
무엇을 실행해야 하는가?

1. 코드 작성: 실제 동작하는 코드의 작성
2. 코드 실행: 실제 동작에 문제없는지 실행 및 확인
3. 코드 개선: 문제가 있는 부분을 발견하고 개선

RAG


RAG Architecture Model
Framework
4
Post Processing
Response
LLM
Client
1
Question
3
Prompt
Semantic
Search
Contextual
Data
2
Vector Database
Original Content
New Content

컨텍스트 윈도우


Input
User Message
Output
Model Response
Input
Previous Turn
Output
Model Response
Input
Previous Turn
Output
Model Response
Context Window
200K tokens

Cursor 에이전트의 흐름

- 듣기: 사용자의 자연어 명령 입력 ("이 함수 리팩토링 해줘")
- 찾기 (RAG): 프로젝트 전체에서 관련된 파일과 문맥 검색
- 컨텍스트: 이전까지의 대화의 맥락 참고하기
- 생각하기 (LLM): 여러 모델을 사용해 수정 계획 수립하기
- 행동하기 (Agent): 여러 파일을 동시에 열어 코드를 수정하기
- 점검: 에러가 없는지 검토 후 적용

AI는 본인이 정답이라고 '주장'하는 결과를 만든다

AI는 '확신에 찬 어조'로 틀린 말을 한다

- AI는 "잘 모르겠습니다"라고 말하지 않는다
- 학습 데이터를 기반으로 '가장 그럴듯한' 답을 생성할 뿐이다
- 이것이 정답인지 아닌지는 AI 스스로 판단하지 못한다

실제로 일어나는 일

- 존재하지 않는 라이브러리를 "설치하세요"라고 안내
- 더 이상 사용하지 않는 문법을 "이렇게 작성하세요"라고 제안
- 우리 프로젝트에 맞지 않는 구조를 "best practice입니다"라며 권장

"AI는 정답을 '아는' 것이 아니라 정답처럼 '보이는' 것을 만들어낸다"

한 단계 만들고 한 단계 검증하기

우리가 해야 할 것

- AI의 결과를 '초안'으로 받아들인다
- 제안된 코드가 실제로 동작하는지 직접 확인한다
- 왜 이렇게 작성했는지 AI에게 다시 물어본다

우리가 로드맵을 작성한 이유

- 개발자는 각 단계를 직접 보며 검증이 가능하지만 우리는 어렵다
- 잘 검증된 로드맵을 만들고, 그 로드맵의 수행이 결과물의 개발로 이어지게 한다.
- 단계를 모두 진행하면 문제없는 결과물이 나오기를 기대한다

최선의 해법: 단계를 가능한 작게 만들고, 각 단계별로 검증한다

컨텍스트가 길면 밟힌다

LLM이 소화할 수 있는 컨텍스트의 크기는 제한이 있다

- 너무 큰 범위의 명령을 입력하면 많은 코드들을 검토해야 한다.
- 너무 오래전 이야기를 기반으로 명령하면 컨텍스트를 잃어버릴 가능성이 높다
- 너무 많은 코드를 작성해야 하면 작성하면서 컨텍스트를 반영하기 어렵다

“한번에 너무 큰 일을 명령하면 실수할 가능성이 높아진다"

우리가 열심히 공부한 이유

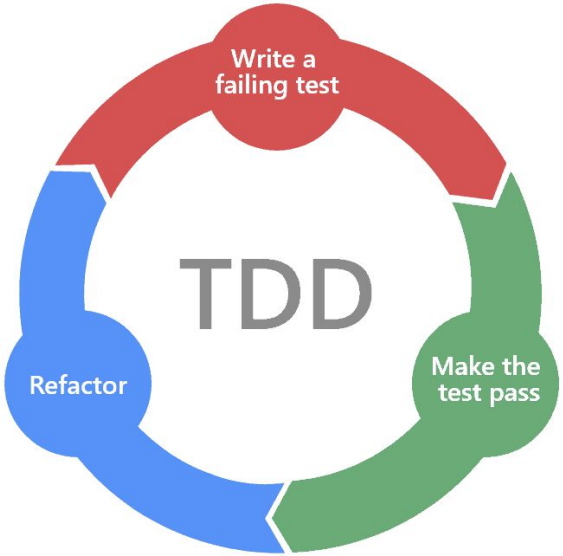
- AI 가 작성한 결과를 100% 신뢰하면 안된다
- 작성한 결과가 기존의 로직에 잘못 연결되어도 AI는 파악하지 못한다
- 이를 위해 결과가 어떻게 동작하는지 확인하고 검증해야 한다.

검증을 위해서 해야하는 것

- 지금 코드는 무엇을 목적으로 작성되었는가
- 지금 코드는 이 목적을 어떻게 달성하였는가
- 지금 코드는 실제로 이 목적을 달성하였는가
- 지금 코드는 문제없이 실행이 되는가

TDD(Test Driven Development)

- 코드를 개발하기 전에 먼저 테스트를 만들고, 그 테스트를 통과할 수 있는 코드를 만드는 것을 목표로 하는 개발 전략
- 개발 전에 목표를 정하고 이 목표를 달성할 때 까지 코드를 지속적으로 개선한다.
- 목적을 달성하라는 추상적인 말 보다 컴퓨터는 명확한 말을 더 신뢰한다
- 이를 위해서 에이전트 코딩에 TDD개념을 활용하면 도움이 된다



질문 1. 코드 실행하기

“XXXX폴더의 모든 로드맵 문서를 검토해 XXX 단계의 목적을 달성하는 코드를 작성해줘.

단순히 해당 단계의 요구사항만 확인하지 말고 문서 전체의 요구사항을 확인한 뒤에 해당 단계의 목적이 무엇인지 명확하게 이해하고 구현해줘. 목적 달성을 위해 안전하지만 효율적인 방식으로 명확하게 수행해줘"

질문 2. 코드 검증하기

“XXXXXX 단계의 요구사항을 달성했는지 실제 테스트를 통해 검증하고 문제가 있는 부분을 확인해 해결해줘.

단순히 해당 단계의 요구사항만 확인하지 말고 문서 전체의 요구사항을 확인한 뒤에 해당 단계의 목적이 무엇인지 명확하게 이해하고 구현해줘. 목적 달성을 위해 안전하지만 효율적인 방식으로 명확하게 수행해줘"

오늘 해결해야 할 문제

1. Cursor IDE를 설치하고 기본 구조(에디터, 채팅, 터미널)를 직접 확인합니다.
2. 앞서 작성한 로드맵의 첫 번째 시나리오를 Cursor에게 명령하여 코드를 생성합니다.
3. 생성된 코드가 시나리오의 요구사항을 충족하는지 테스트로 검증합니다.
4. 검증 과정에서 발생한 문제를 AI와 함께 해결하고, 해결 과정을 기록합니다.
5. (선택) 완성된 코드를 GitHub에 커밋합니다.

Q&A