바이브 코딩 실전 아키텍처 ②

Windows · macOS · Linux 데스크톱 앱 만들기

네이티브 · Qt, Python Flet, Electron — 무엇으로, 얼마나 어렵게, 어디에 쓰나

C# · WinUI · WPF | Swift · SwiftUI | GTK | Qt · PySide6 | Flet · Flutter | Electron

AI 애플리케이션 개발 과정 · 보충 강의

오늘의 흐름

OS에서 시작해 세 가지 방식으로

01	**OS별 프레임워크**	Windows · macOS · Linux가 기본으로 주는 UI 도구와 라이브러리
02	**네이티브 · Qt**	C# · Swift로 OS마다, 또는 Qt로 C++ · Python 한 소스
03	**Python Flet**	Flutter 엔진 위에서 Python 한 소스로 모든 OS
04	**Electron**	Chromium + Node.js로 웹 기술 그대로 데스크톱 앱
05	**비교 · 선택 · 배포**	난이도, 실제 사용 예, 선택 기준, 그리고 웹 개발과의 차이

01 · OS별 프레임워크

웹이 있는데 왜 데스크톱 앱인가

하드웨어 접근

시리얼 · USB 장비, PLC, 바코드 스캐너, 로컬 GPU

오프라인 · 폐쇄망

인터넷이 끊긴 공장 · 보안 구역에서도 동작

로컬 파일 · 대용량

수 GB 로그 · 영상 · 모델 파일을 업로드 없이 바로 처리

OS 통합

트레이 아이콘, 전역 단축키, 시작 프로그램, 알림

반응성

네트워크 왕복 없이 즉시 반응하는 화면

대신 생기는 숙제

OS마다 빌드 · 설치 파일 · 서명 · 업데이트

핵심 질문: **세 OS에서 같은 앱을, 얼마나 적은 노력으로** 만들고 유지할 것인가.

01 · OS별 프레임워크

OS가 기본으로 주는 UI 도구

구분	Windows	macOS	Linux
주 언어	C# (.NET), C++	Swift, Objective-C	C, C++, Python, Rust
최신 UI	WinUI 3 (Windows App SDK)	SwiftUI	GTK 4 (GNOME) · Qt 6 (KDE)
전통 UI	WPF, WinForms, Win32	AppKit (Cocoa)	GTK 3, X11 · Wayland 직접
개발 도구	Visual Studio	Xcode (Mac 필수)	GNOME Builder, Qt Creator
배포 형식	.exe · .msi · MSIX, MS Store	.app · .dmg · .pkg, App Store	.deb · .rpm · AppImage · Flatpak · Snap
서명	코드 서명 인증서 (SmartScreen)	Developer ID + 공증(Notarization)	저장소 · 패키지 서명 (선택적)

세 열에 **겹치는 칸이 하나도 없다** — 그래서 크로스플랫폼 도구가 생겼다.

01 · 라이브러리

앱을 만들 때 실제로 쓰는 라이브러리

용도	C# · .NET	Swift	Qt (C++ · PySide6)	Electron (Node)
UI 구조	XAML + MVVM (CommunityToolkit.Mvvm)	SwiftUI 상태(@State · @Observable)	Qt Widgets 또는 QML	React · Vue 등 웹 프레임워크
차트	LiveCharts2, ScottPlot	Swift Charts	Qt Graphs, pyqtgraph	ECharts, Chart.js
시리얼 · 장비	System.IO.Ports	IOKit, ORSSerialPort	Qt Serial Port	serialport (npm)
로컬 DB	EF Core + SQLite	SwiftData, Core Data	Qt SQL, sqlite3	better-sqlite3
웹뷰 내장	WebView2 (Edge)	WKWebView	Qt WebEngine	기본이 Chromium
패키징 · 업데이트	MSIX, Velopack	Xcode Archive, Sparkle	windeployqt · macdeployqt, PyInstaller	electron-builder, Forge, electron-updater

Flet은 Python 생태계(pandas · requests 등)를 그대로 쓰고, UI 확장은 flet-* 패키지로 추가한다.

02 · 세 가지 접근법

세 방식의 층 구조: 누가 화면을 그리나

A · 네이티브 / Qt

앱 코드 · C# / Swift / C++ · Python

UI 툴킷 · WinUI · SwiftUI · GTK / Qt

OS 그래픽 API · 직접 호출

OS · Windows / macOS / Linux

화면: **OS 컨트롤** (Qt는 OS 모양을 흉내 내 직접 그림)

OS별 소스 · C#·Swift는 OS마다 따로, Qt는 하나

B · Python Flet

앱 코드 · Python

Flet 런타임 · Python 프로세스

Flutter 엔진 · Dart, 자체 렌더링

OS · 창 하나와 GPU만 빌림

화면: **Flutter가 픽셀을 직접** — 모든 OS에서 같은 모양

OS별 소스 · 하나 (+ 모바일 · 웹)

C · Electron

앱 코드 · HTML · CSS · JS/TS

Chromium · 화면 (렌더러)

Node.js · 파일 · OS 기능 (메인)

OS · 브라우저 엔진을 통째로 동봉

화면: **내장 브라우저가 웹페이지로**

OS별 소스 · 하나 (+ 웹 재사용)

같은 “측정” 버튼, 세 가지 코드

C# · WPF

Windows 전용

```
<!-- MainWindow.xaml -->
<StackPanel>
  <TextBlock x:Name="Temp"/>
  <Button Content="측정"
    Click="OnMeasure"/>
</StackPanel>

// MainWindow.xaml.cs
void OnMeasure(object s,
  RoutedEventArgs e) =>
  Temp.Text = $"{Read()} ℃";
```

Swift · SwiftUI

macOS (+ iOS)

```
struct ContentView: View {
  @State var temp = 0.0

  var body: some View {
    VStack {
      Text("\(temp) ℃")
      Button("측정") {
        temp = read()
      }
    }
  }
}
```

Python · PySide6 (Qt)

Windows · macOS · Linux

```
from PySide6.QtWidgets import (
  QApplication, QWidget,
  QVBoxLayout, QLabel, QPushButton)

app = QApplication([])
w = QWidget(); lay = QVBoxLayout(w)
lbl = QLabel("-")
btn = QPushButton("측정")
btn.clicked.connect(
  lambda: lbl.setText(f"{read()} ℃"))
lay.addWidget(lbl); lay.addWidget(btn)
w.show(); app.exec()
```

C# · Swift는 **OS마다 다시 작성**, Qt는 **한 소스를 OS마다 빌드**한다.

네이티브 vs Qt: 장단점과 주의점

OS 네이티브 (C# · Swift)

장점 OS와 똑같은 모양 · 동작, 최신 OS 기능을 가장 먼저, 가볍고 빠름

단점 OS 수만큼 코드 · 언어 · 개발 도구, Mac 앱은 Mac에서만 빌드

적합 한 OS만 지원하는 사내 도구, 고품질 상용 앱

C#의 크로스 선택지 Avalonia(데스크톱 3종), .NET MAUI(Windows · macOS · 모바일, Linux 공식 미지원)

Qt (C++ · PySide6)

장점 한 소스로 3개 OS + 임베디드, 30년 가까이 검증, 산업 · 계측 분야 표준급

단점 위젯 모양이 다소 고전적, C++ 학습 곡선, 배포 크기 수십 MB

적합 장비 제어 HMI, 계측 · 영상 도구, 장기 유지보수 제품

Widgets vs QML 업무용 폼은 Widgets, 터치 · 애니메이션 UI는 QML

라이선스 주의 Qt와 PySide6는 LGPL(동적 링크 시 상용 배포 가능), PyQt6는 GPL 또는 상용 라이선스 — 판매용 앱이면 배포 전에 반드시 확인

Flet 구조: Python이 지시하고 Flutter가 그린다

main.py
내 코드 · pandas, requests, 시리얼 등 Python 패키지

→

Flet 런타임
컨트롤 트리 보관, 바뀐 부분만 계산 (Python 쪽)

변경분 →
← 이벤트

Flutter 클라이언트
Dart 엔진이 GPU로 위젯을 직접 렌더링 (화면 쪽)

한 소스 → flet build <대상>

windows	macos	linux	apk / aab	ipa	web
.exe 폴더	.app 번들	실행 번들	Android	iOS	정적 웹앱

패키징된 앱에는 Python 런타임이 함께 들어가 사용자가 Python을 설치할 필요가 없다. 단, **각 OS용 빌드는 그 OS에서** 실행해야 한다.

Python 한 파일로 세 OS 앱

```python
import flet as ft

def main(page: ft.Page):
    page.title = "설비 모니터"
    temp = ft.Text("-", size=40)

    def on_measure(e):
        temp.value = f"{read()} ℃"
        page.update()

    page.add(temp,
        ft.Button("측정", on_click=on_measure))

ft.run(main)
```

```
$ uv add flet
$ flet run              # 데스크톱 창으로 실행
$ flet build windows    # 배포용 빌드
```

난이도: 낮음

Python만 알면 된다. HTML · CSS · Dart 불필요

강점

Material 디자인 기본 제공, AI · 데이터 코드와 한 언어, 모바일까지 확장

약점

생태계가 젊다, 네이티브 확장 패키지는 모바일 빌드에서 제약, OS 고유 모양은 아님

버전 메모

1.0 기준 진입점은 ft.run(main), 구버전 예제는 ft.app(target=main)

Electron 구조: 프로세스 두 개와 그 사이의 다리

메인 프로세스 · 1개

Node.js

- 창 생성 · 메뉴 · 트레이
- 파일 시스템, 시리얼, DB
- 자동 업데이트

preload.js

IPC 다리

contextBridge로 허용한 함수만 화면에 노출

렌더러 프로세스 · 창마다

Chromium

- HTML · CSS · JS 화면
- React · Vue 그대로 사용
- Node 직접 접근은 차단

Python 로직을 붙이려면

FastAPI를 PyInstaller로 실행 파일화 → 메인 프로세스가 사이드카로 실행 → 렌더러는 localhost API 호출

대가

앱마다 Chromium을 통째로 동봉 → 설치 크기 약 100MB 안팎, 메모리 사용량이 큼

04 · C · Electron

같은 기능을 세 파일로

main.js — 메인 프로세스

```
const { app, BrowserWindow,
  ipcMain } = require("electron");
const path = require("node:path");

ipcMain.handle("read-temp",
  () ⇒ readTemp());

app.whenReady().then(() ⇒ {
  const win = new BrowserWindow({
    webPreferences: { preload:
      path.join(__dirname,
        "preload.js") } });
  win.loadFile("index.html");
});
```

preload.js — 다리

```
const { contextBridge, ipcRenderer }
  = require("electron");
contextBridge.exposeInMainWorld("api", {
  readTemp: () ⇒ ipcRenderer.invoke("read-temp"),
});
```

renderer.js — 화면 (index.html에서 로드)

```
btn.onclick = async () ⇒ {
  temp.textContent =
    `${await window.api.readTemp()} ℃`;
};
```

난이도: 중간 — 웹 개발자에겐 쉽지만 프로세스 · IPC · 보안 설정을 이해해야 한다. 배포는 electron-builder 또는 Electron Forge.

05 · 비교

한 표로 보는 네 가지 선택지

항목	네이티브 C# · Swift	Qt · PySide6	Python Flet	Electron
언어	OS마다 다름	C++ 또는 Python	Python	JS · TS + HTML · CSS
3개 OS 소스	3벌	1벌	1벌 (+모바일 · 웹)	1벌 (+웹 재사용)
화면 모양	OS 그대로	OS 유사	Material, OS 공통	웹 디자인 자유
설치 크기 (대략)	가장 작음	수십 MB	수십 MB	100MB 안팎
메모리 · 속도	가장 가벼움	가벼움	보통	무거움
OS 기능 접근	전부, 최신 기능 즉시	대부분 (모듈 풍부)	Python 패키지 경유	Node 모듈 경유
학습 난이도	높음 (×OS 수)	중상 (C++) · 중 (Python)	낮음	중 (웹 경험 시 낮음)
AI 코드 생성	자료 많음	자료 많음	버전 차이 주의	자료 가장 많음

05 · 비교

난이도와 실제 사용 예

네이티브

난이도 높음 · 4/4

Windows · macOS 기본 앱들, Paint.NET(C#), Apple의 Xcode · Safari

Qt

난이도 중상 · 3/4

OBS Studio, VLC, Telegram Desktop, KDE Plasma, 다수의 계측 · 장비 HMI

Flet

난이도 낮음 · 1/4

사내 데이터 대시보드, AI 챗 · 문서 도구, 설비 모니터 PoC, 교육 · 내부 업무 앱

Electron

난이도 중간 · 2/4

VS Code, Slack, Discord, Obsidian, Notion · Figma 데스크톱

대형 상용 서비스는 **Electron**, 장비 · 미디어는 **Qt**, Python 팀의 빠른 내부 도구는 **Flet**이 강하다.

05 · 선택 기준

네 개의 질문으로 고르기

Q1 한 OS만 지원하고, 그 OS다운 품질이 중요한가? — 예 → **네이티브 C# · Swift**

↓ 아니오

Q2 장비 제어 · 영상처럼 성능과 장기 유지보수가 핵심인가? — 예 → **Qt (Python이면 PySide6)**

↓ 아니오

Q3 이미 웹 프런트엔드 코드 · 웹 개발자가 있는가? — 예 → **Electron (가볍게: Tauri)**

↓ 아니오

Q4 Python 중심이고, 빠르게 · 모바일까지 가고 싶은가? — 예 → **Python Flet**

05 · 배포

소스는 하나여도 빌드는 OS마다

CI 빌드 매트릭스 — 세 OS 러너에서 동시에 (Flet 예)

```yaml
jobs:
  build:
    strategy:
      matrix:
        include:
          - { os: windows-latest, target: windows }
          - { os: macos-latest,   target: macos }
          - { os: ubuntu-latest,  target: linux }
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v4
      - run: pip install flet
      - run: flet build ${{ matrix.target }}
```

Linux 러너에는 GTK 등 빌드용 시스템 패키지 설치 단계를 추가한다

서명

서명 없는 Windows 앱은 SmartScreen 경고, macOS 앱은 공증 없이는 실행이 막힌다

설치 파일

빌드 결과물을 Inno Setup · MSIX, .dmg, AppImage 등으로 한 번 더 포장

자동 업데이트

Electron은 electron-updater, 그 외는 Velopack · Sparkle 또는 사내 다운로드 서버

폐쇄망

설치 파일을 오프라인으로 배포하고 버전 확인 API만 사내 서버에

06 · 웹 개발과의 차이

일반 웹 개발과 무엇이 다른가

항목	웹 (지난 시간 구성)	데스크톱 앱
실행 위치	서버에서 실행, 브라우저는 화면만	사용자 PC에서 실행
배포 · 업데이트	서버에 한 번 배포하면 모두에게 즉시	OS별 빌드 · 서명 · 설치, 업데이트는 사용자 PC마다
버전 관리	모두가 최신 버전 하나	여러 버전이 동시에 현장에 존재
로컬 자원	브라우저 샌드박스 안에서 제한적	파일 · 장비 · OS 기능 자유롭게
오프라인	대부분 불가 (PWA로 일부)	기본적으로 가능
보안 경계	비밀값은 서버에만 두면 안전	앱 안의 모든 것은 사용자 손에 — 키를 넣으면 노출
오류 확인	서버 로그 · Grafana로 한곳에서	사용자 PC에 흩어짐 → 크래시 리포트 수집 필요

둘은 이어진다: 데스크톱 앱이 프런트엔드가 되어 지난 시간의 FastAPI 백엔드를 호출하는 구성이 흔하다.

정리

중간 층이 두꺼울수록 소스는 하나로, 앱은 무겁게

1 **품질 · 성능 우선**
네이티브 또는 Qt

2 **웹 자산 재사용**
Electron

3 **Python으로 빠르게**
Flet — 이 과정의 출발점

프롬프트 예시

“Flet 1.0(ft.run 방식)으로 Windows · macOS · Linux 데스크톱 앱을 만들어 줘. 화면은 설비 목록과 실시간 온도 차트, 데이터는 기존 FastAPI 백엔드의 /equipment API에서 받는다. API 키는 앱에 넣지 말고, 세 OS용 flet build를 GitHub Actions 매트릭스로 구성해 줘.”