https://youtu.be/tJkD0t0GaMk?list=TLGGqfZxGbwsXKMyNjEyMjAyNQ

https://www.youtube.com/watch?v=wmbq6fa7pE8&list=PLIAzIZzCjtLJQzoK-1n6R_KVTXm4tHZqP&index=13

這份教學原始碼主要探討**全端應用程式開發**的實務流程，強調如何整合**第三方 API**（如 Google Drive）來建立功能完整的作品展示平台。講師主張在 AI 輔助開發的時代，應採取**計畫驅動**而非單純規格驅動的模式，透過與 AI 進行**分階段的對話**，從 UIUX 原型設計逐步進展至後端邏輯與資料庫管理。文中詳盡說明了 **Git 版本控制**的進階操作，特別是如何利用 **Rebase** 維護乾淨的提交歷史，並教導學生在遇到合併衝突時保持冷靜，透過理解工作區與儲存庫的關係來解決問題。最後，教學強調了**程式碼整潔與模組化**的重要性，建議開發者應適時將複雜邏輯抽離成 **React Hooks**，並透過持續的測試與代碼審查，建構出高效且易於維護的現代化網路應用

這份教學資源記錄了 NTU 網頁程式設計課程中，關於**全端應用程式開發與第三方 API 整合**的技術細節。內容詳盡探討了 **Git 版本控制**中的合併衝突與標籤使用，並強調在架構設計時應遵循「**單一真理來源 (Single Source of Truth)**」原則，以避免資料庫與靜態檔案間的同步錯誤。講者進一步說明了 **Express 後端實作**、**RESTful API 設計**以及**關聯式資料庫 (SQLite)** 的核心觀念，特別是索引（Indexing）對提升查詢效率的重要性。此外，針對 **Google Drive 與 Map API 的串聯**，文中分享了處理權限驗證與 **速率限制 (Rate Limiting)** 的實務經驗，並展示如何利用 **AI 輔助開發**來優化程式碼重構與自動生成技術文件。

Discuss 全端應用程式開發.

全端應用程式開發（Full-Stack App Development）是指同時涵蓋**前端（Frontend）**使用者介面與**後端（Backend）**伺服器、資料庫邏輯的軟體開發過程,。相較於純前端開發，全端開發的複雜度顯著提高，開發者需要整合多種技術組件，並透過外部 API 擴充功能，以建構出完整且合理的網路應用。

以下根據來源內容，探討全端開發的核心架構、現代開發流程及關鍵技術觀念：

1. 全端開發的技術架構

一個典型的全端應用程式通常包含以下層次,：

• **前端（Frontend）：** 負責與使用者互動，常見技術如 **React**。在全端架構中，可能會同時存在「使用者端」與「管理員端」兩個前端介面，它們共用同一個後端伺服器。

• **後端（Backend）：** 處理商務邏輯與路由（Routing）。來源中建議使用 **Node.js** 作為執行環境，並搭配 **Express** 框架來快速建立 API 服務,,。

• **資料庫（Database）：** 負責結構化資料的存取與管理。

  ◦**SQLite**：適合輕量級應用或開發初期使用，因其易於部署且不需複雜設定,,。

  ◦**PostgreSQL/MySQL**：適合更具規模、高併發需求的應用,,。

• **儲存空間（Storage）：** 存放非結構化大型檔案（如圖片、影片）。為了節省成本與簡化管理，開發者常會串接 **3rd-party APIs**，例如利用 **Google Drive** 提供儲存容量,,。

• **溝通橋樑：** 前後端透過 **HTTP 請求**（常搭配 Axios 庫）進行通訊，並利用 **JWT（JSON Web Token）** 進行身分驗證，確保資料安全,。

2. 現代 AI 輔助開發流程

在 AI 工具盛行的背景下，全端開發的模式已從傳統的「寫程式（Coding）」轉向「溝通與規劃」,。

• **計畫驅動（Plan-driven）而非規格驅動（Spec-driven）：** 來源建議不要強求一次寫完所有細節（Spec），而是與 AI 溝通出一份**開發計畫（Implementation Plan, PL）**，將複雜的 App 拆解成多個階段（如：原型製作、資料庫規劃、API 串接等）獨立驗收,,,。

• **開發效率的飛躍：** 以前需要耗時 17 到 24 天的開發工作，現在透過 AI 輔助，可能在一個下午內就能完成大部分雛形,。

• **AI 的角色：** AI 可以生成程式碼、自動編寫文件與測試腳本（Test Scripts）,。然而，開發者仍需具備技術底子，進行 **Code Review**，並在 AI 遺漏 Best Practice（如 React 的 Custom Hooks）時手動介入修正,,。

3. 關鍵開發原則與挑戰

• **單一事實來源（Single Source of Truth）：** 這是資訊工程的核心原則。資料應保存在單一位置（如資料庫），避免在多處存放副本，以免造成資訊不同步,。

• **資料庫設計與索引（Indexing）：** 資料庫設計需遵循 **ACID 原則**（原子性、一致性、隔離性、持久性）。當資料量大時，必須建立**索引（Index）**，將搜尋時間從線性時間 O(n) 降至對數時間 O(logn)，以維持系統效能,。

• **處理外部 API 限制：** 串接第三方服務（如 Google Drive）時，常會遇到 **Rate Limit（429 Too Many Requests）**。此時需加入 **Middleware（中介軟體）** 或代理節點（Proxy）來管理請求頻率，而非直接讓前端頻繁存取,。

• **版本控制（Git）：** 在全端協作中，Git 是不可或缺的工具。遇到 **Merge Conflict（合併衝突）** 時不應恐慌，這只是提醒開發者需手動檢閱不同版本間的改動，而非程式碼出錯,,。

總結與類比

全端開發就像是**蓋一棟房子**：前端是精美的裝潢與門窗（使用者看的見的部分），後端則是支撐房屋的水電管道與樑柱（看不見的邏輯），而資料庫就是地下室的儲藏室。

為了優化存取效率，可以將資料存取行為類比為「尋找鑰匙」：

• **快取（Cache）：** 鑰匙就在**口袋**裡，拿取最快。

• **記憶體（Memory）：** 鑰匙在房間的**抽屜**裡，需要站起來走過去拿。

• **硬碟（Hard Drive）：** 鑰匙在**整棟房子的某個角落**，如果沒有索引標籤，你可能得翻遍全家才能找到,。

為什麼 Git 的 rebase 指令比 merge 更能維持提交歷史的整潔？

在開發複雜的全端應用程式時，老師建議如何與 AI 溝通？

關聯式資料庫設計中，Primary Key 與 Foreign Key 的功用為何？

Discuss Git 版本控制.

Git 版本控制在現代開發中扮演著至關重要的角色。根據來源，Git 不僅是與他人協作時避免溝通不良和程式碼衝突的必備工具，即使是個人開發，養成使用 Git 進行備份與版本管理的習慣也大有裨益。

以下根據來源整理 Git 版本控制的核心觀念、操作流程與衝突解決方式：

1. Git 的核心角色與結構

理解 Git 最重要的是搞清楚以下三個角色的關係：

• **GitHub (Remote Repository)：** 遠端的版本控制中心。

• **Local Repository：** 你電腦本地端的版本控制中心，負責儲存每一版的紀錄。

• **Working Area (Working Directory)：** 你實際撰寫、修改程式碼的地方。

在將程式碼從工作區存入本地倉庫之前，還會有一個 **Staging Area (暫存區)**，透過 `git add` 將修改加入此區，再經由 `git commit` 正式提交到本地倉庫。

2. 重要指令與同步機制

來源詳細說明了如何讓本地與遠端保持同步：

• **取得更新：**

◦ **Fetch：** 僅將程式碼從遠端拉到本地倉庫，不會影響目前的工作區，因此**執行 fetch 時不會產生衝突 (Conflict)**。

◦ **Pull：** 等於 **Fetch + Merge**，除了拉下程式碼外，還會自動嘗試將遠端程式碼合併到目前的工作區，這時才可能產生衝突。

• **推送更新：** 使用 `git push` 將本地的更新同步回遠端倉庫。

• **分支管理：** 透過 `git checkout` 切換不同的分支（Branch），讓開發者可以進行平行開發與管理。

• **標籤 (Tag)：** 用於標記特定的版本（例如發布版）。雖然可以切換回標籤所在的版本觀看程式碼，但不能直接在該狀態下修改提交，若要修改則需另開分支。

3. 合併 (Merge) 與 衍合 (Rebase)

當需要整合不同分支的程式碼時，來源比較了兩種常見做法：

• **Merge：** 將程式碼合併，但在 Git History（歷史紀錄）上會留下合併的節點，有時會顯得混亂。

• **Rebase：** 意即「重新定義基準」。它會將你目前的修改暫時移開，把基準點更新到最新版本後，再將你的修改疊加回新基準點之上。

• **建議：** 使用 **Rebase** 能讓歷史紀錄保持「乾淨」，避免不同開發者的紀錄交織在一起，方便日後追蹤或回溯。

4. 處理衝突 (Merge Conflict)

來源強調，**衝突並不是錯誤 (Bug)**，而是因為兩個檔案在同一個地方有不同的修改，系統無法自動決定保留哪一個，所以需要人工介入。

• **衝突發生的時機：** 通常發生在 `git push` 時，如果遠端已有他人更新，你的推送會被拒絕。

• **解決流程：**

1. 先執行 `fetch` 拿到遠端最新版，再進行 `rebase`。
2. 打開有衝突的檔案，Git 會用箭頭與等號標註出「目前變更」與「來源變更」。
3. **手動決定**要保留哪些內容後，再次執行 `add`、`commit` 並接續 `rebase --continue` 即可完成。

• **小秘訣：** 遇到衝突不要慌張，可以利用程式編輯器（如 VS Code）的顏色標示功能來協助處理，或請教 AI 尋求解決指令，千萬不要隨意刪除檔案。

5. 遠端名稱的定義 (Origin vs. Upstream)

在協作開發中，通常會設定兩個遠端指標：

• **Origin：** 指向你自己 Fork 出來的遠端倉庫。

• **Upstream：** 指向原始的（別人的）專案倉庫。當原作者持續改版時，你需要從 Upstream 同步最新程式碼到本地。

比喻理解： 我們可以把 Git 想像成一個**「具備時光機功能的圖書館」**：你的工作電腦是**「書桌」**，你在這寫書；**「本地倉庫」**是你的**「私人保險箱」**，每寫完一個章節就鎖進去一份副本；而** **則是**「公共總圖書館」**。

當你寫完想放進圖書館（`push`）時，如果發現別人已經先放了一本新版在書架上，圖書館員會拒絕你。這時你得先去看看別人寫了什麼（`fetch`），把你的內容重新整理（`rebase`），確保跟最新的劇情接得上，最後才能順利上架。

比較 Git 指令 fetch 與 pull 的主要差異。

說明 Git rebase 相較於 merge 的優點。

解釋資料庫設計中「單一事實來源」原則的重要性。

Discuss 第三方 API 整合.

在現代全端開發中，**整合第三方 API（3rd-party APIs）** 是建構功能完備且合理應用程式的關鍵，其核心目的在於**避免「重複打造輪子」（Reinventing the wheel）**。開發者可以透過付費或免費串接外部成熟的服務，顯著擴展應用程式的功能邊界。

以下根據來源內容，討論第三方 API 整合的策略、流程與常見挑戰：

1. 為什麼要整合第三方 API？

• **功能擴充與專業化：** 對於個人或小團隊而言，自行開發複雜的地理位置、儲存或支付系統非常困難。例如，利用 **Google Maps API** 可以快速實作旅遊規劃（Travel Planner）或位置相關應用；利用 **Google Drive API** 則能解決大量圖片或檔案的儲存需求。

• **提升開發效率：** 傳統開發可能需要數週才能完成的功能，透過 API 與現代 AI 工具輔助，可能在一個下午內就達成初步整合。

2. 整合流程與最佳實踐

整合第三方 API 並非單純的代碼撰寫，而是一套結構化的流程：

• **優先處理高風險部分：** 在開發計畫（Implementation Plan）中，API 串接通常被視為**不確定性較高**且**風險較大**的部分，應在初期階段優先處理與測試。

• **Google Cloud Console 服務設定：** 以 Google 服務為例，開發者需在雲端後台建立專案、開啟特定的 API 功能（如 Google Drive API），並配置**憑證（Credentials）**、**金鑰（Keys）****服務帳號（Service Account）**。

• **身份驗證機制：** 串接時常涉及 **OAuth 2.0** 驗證，需區分「服務帳號」與「使用者帳號」的權限差異，以確保伺服器能正確且安全地存取資源。

• **自動化測試：** 建議使用 AI 生成**測試腳本（Testing Scripts）**，直接對 API 端點進行測試，透過 log 紀錄（如紅黃綠燈號）快速診斷連線或權限問題。

3. 面臨的挑戰與解決方案

• **速率限制（Rate Limit）：** 第三方服務為了防禦攻擊或節省資源，通常會限制每秒請求數。若請求過於頻繁，會遇到 **429 Too Many Requests** 錯誤。

  ◦ *解決方案：* 來源建議不要讓前端直接打 API，而是在後端加入一個**代理端點（Proxy）**或**中介軟體（Middleware）**來統一管理與轉發請求，並實作重試（Retry）機制。

• **資料一致性（Single Source of Truth）：** 在整合 API 時，需注意資料不應在多處存放副本，以免產生資訊不同步的「悲劇」。

• **API 規格更迭：** 整合時可能會遇到文件版本不符或方法（Method）不存在的問題，此時需細心檢閱 AI 生成的代碼是否符合最新的 API 規範。

4. AI 在 API 整合中的角色

在整合過程中，AI 扮演了「技術文件助手」與「代碼生成器」的角色。它能幫開發者快速寫出繁瑣的 **Integration Guide**（整合指南），並處理大量具備 **hyperlink** 的文件結構調整。對於開發者來說，重點在於具備技術底子，以便在 AI 遺漏 Best Practice 或產生錯誤邏輯時進行手動干預。

類比理解： 第三方 API 整合就像是**為你的餐廳外包「外送服務」或「食材供應」**。你不需要自己養一支外送車隊（自己寫 Google Maps），也不需要自己開農場（自己蓋大型儲存伺服器），你只需要與專業供應商簽好合約（設定憑證與 API Key），並設定好接收窗口（後端 Routing），就能讓你的餐廳服務變得非常強大。但你必須隨時注意供應商的供貨速度（Rate Limit），並確保訂單資訊（Data）在雙方系統中是一致的。

Git fetch 與 git pull 的主要差異為何？

為何使用 rebase 管理提交歷史會比 merge 乾淨？

在資料庫設計中，「單一真理來源」原則有何重要性？

Discuss React 組件設計.

在現代全端開發中，**React** 是建構使用者介面（Frontend）的核心技術。根據來源內容，React 組件設計不應只是撰寫程式碼，而應遵循特定的**最佳實踐（Best Practices）**，並善用 AI 工具來優化結構與效能。

以下是 React 組件設計的關鍵面向：

1. 邏輯與版面分離：使用 Custom Hooks

在設計組件時，最理想的做法是將**狀態操作（State Operations）**與**視覺佈局（Layout）**分開。

• 避免混合： 如果將所有功能函數直接寫在組件物件中，會導致程式碼過於臃腫且難以維護。

• 封裝 Hook： 建議將關於 state 的處理邏輯提取（Refactor）到 **Custom Hooks** 中，讓組件專注於渲染介面。這樣能使組件結構更清晰，也便於邏輯的重用。

2. 組件的拆解與重構

當應用程式變得複雜時，組件的體積往往會迅速增長。

• 手動介入拆分： 雖然 AI 可以協助生成組件，但它通常會從簡單寫到複雜，且不一定會主動拆分過大的程式碼塊。開發者需要具備「代碼潔癖」，主動指示 AI 將龐大的組件拆解成更小、更專一的子組件。

• 階層管理： 在設計時需考量資料的依賴關係。如果組件階層較淺（例如只有兩層），直接透過 Props 傳遞資料通常比使用 **Context API** 更簡潔，能避免增加不必要的複雜度。

3. 效能優化與狀態緩存

為了建構高效的應用，組件設計需考量運算成本：

• useMemo 的應用： 對於耗時的計算邏輯，應使用 `useMemo` 來緩存結果。這就像是「背答案」一樣，當相同的輸入再次發生時，組件可以直接回傳緩存值，而不必重新計算，從而提升系統反應速度。

4. AI 輔助 UI/UX 設計

現代開發流程中，AI 能極大地加速組件的視覺設計：

• 遵循通用慣例（Common Practices）： 對於常見的 UI 組件（如彈出視窗 Popup、功能按鈕），AI 已經掌握了成熟的設計模式，開發者只需描述功能需求，不需從零開始調整像素。

• 自動配色與佈局： 開發者可以提供一個底色，請 AI 根據該色彩與特定風格進行自動配色，這比手動調整外觀更有效率。

• 快速原型： 透過簡單的草圖或描述，AI 可以在一個下午內完成以前需要數週才能開發出的組件原型。

5. 安全與權限設計

在全端架構下，組件設計還需考量身分驗證。例如在管理員端的組件中，通常會整合 **JWT（JSON Web Token）** 進行登入管理，確保敏感的操作（如新增、修改、移除畫作）僅限授權使用者存取。

比喻理解： React 組件設計就像是**組裝一套模組化家具**。一個好的組件應該像是一個功能獨立的零件（如一個抽屜），它的**內部滑軌邏輯（Custom Hooks）**把手與面板（Layout）**。如果你把所有東西都焊接在一起，當家具變大時就會變得很難搬動；適時地將它拆解成小零件，這套家具才能靈活地適應不同的房間空間。

比較 Git 指令 fetch 與 pull 的差異。

說明 Git 中 rebase 與 merge 的優缺點。

簡述在 Web 開發中建立資料庫索引的目的。

Discuss AI 輔助開發策略.

在 AI 輔助開發的時代，軟體開發的重心已從單純的「撰寫程式碼」轉向**「溝通、規劃與審核」**。根據來源，現代開發者應將自然語言視為一種新的程式語言，透過有效的策略引導 AI 建構複雜的全端應用。

以下是來源中提到的 **AI 輔助開發策略**：

1. 計畫驅動（Plan-driven）而非規格驅動（Spec-driven）

來源指出，雖然有一派開發者提倡將厚重的規格書（Spec）丟給 AI 讓其自動執行數小時，但更好的做法是與 AI 共同制定**開發計畫（Implementation Plan, PL）**。

• 分批次溝通： 由於人類很難在一開始就想清楚所有細節，建議採取「多輪對話」，分階段與 AI 溝通。

• 避免 AI 「腦補」： 若一次給予太多模糊需求，AI 會自行填補細節，導致最終成果偏離預期。最好的方式是**一個步驟完成後，經驗收再進行下一步**。

2. 優先處理高風險與不確定性

在制定計畫時，應優先處理**技術風險高或不確定性大**的部分。

• API 串接優先： 例如在開發全端應用時，Google Drive 或 Google Maps 的 API 串接是不確定的風險點，應在初期先行驗證其可行性與權限設定，而非最後才做。

• 快速原型製作： 以往需要 17 到 24 天才能完成的開發工作，現在透過 AI 輔助，一個下午就能做出可運作的原型。來源建議直接請 AI 做出「真的網頁」進行測試，而非使用傳統的紙本原型。

3. 自動化測試與文件並行

AI 的強項在於處理繁瑣的文字與腳本工作，開發者應善加利用：

• 階段性驗收： 每個階段完成後，請 AI 撰寫**測試腳本（Testing Scripts）**並執行，確保功能正確。

• 文件即時更新： 請 AI 針對目前的路由（Routing）或架構撰寫文件。當檔案目錄結構變動時，AI 能高效地修正所有文件中失效的**超連結（Hyperlinks）**，這對人類來說是極其耗時的工作。

4. 開發者的「程式碼潔癖」與手動介入

雖然 AI 能生成大量程式碼，但開發者仍需維持**程式碼品質與架構**：

• **主動重構（Refactoring）：** AI 寫程式通常是從簡單疊加到複雜，往往不會主動拆分過大的程式碼區塊。開發者需具備「潔癖」，主動指示 AI 將大型組件拆解成多個小組件，或將狀態邏輯提取為 **Custom Hooks**。

• **遵循 Best Practice：** 對於 UI/UX 的通用慣例（如 Popup 視窗、自動配色），可以直接交給 AI 處理；但對於特定的商業邏輯或權限管理（如 JWT），開發者需仔細審閱（Code Review）。

5. 管理 AI 的上下文空間（Context Window）

開發者必須意識到 AI 的「記憶」有限：

• **Token 限制：** 在同一個對話框（Chat）中，隨著對話變長，早期的細節會被壓縮或遺忘。

• **適時更換對話框：** 來源建議不要一個對話框用到底，應在完成一個大階段後，開新的對話框並餵入最新的程式碼，以確保 AI 掌握最精確的上下文資訊。

比喻理解： AI 輔助開發就像是**「指揮一間高效能的建築設計事務所」**。你（開發者）是總建築師，不需要親手搬磚頭（寫每一行 C），但你必須畫出清晰的施工圖（開發計畫），並在每一層樓蓋好時進行驗收（自動化測試）。如果發現事務所把房間規劃得太亂，你必須要求重新裝潢（重構）。如果你一直跟同一個助理講話講太久，他可能會開始忘記你第一層樓要用什麼建材（Token 限制），這時你就需要換一個清醒的助理（開新對話框）並給他看目前的施工進度。

Git 中的 primary key 與 foreign key 有何區別？

為什麼在開發畫廊 App 時選擇 SQLite 而非 PostgreSQL？

在整合 Google Drive API 時，如何解決 429 錯誤？

Discuss Git 版本管理.

根據來源內容，Git 版本管理是全端開發中不可或缺的工具。它不僅能幫助個人開發者進行**備份與版本控制**，更是多人協作時避免溝通不良、防止程式碼衝突（Conflict）的關鍵技術。

以下為 Git 版本管理的詳細探討：

1. 核心架構與角色

理解 Git 的運作需先釐清四個主要角色與區域的關係：

• **遠端倉庫（Remote Repository，如 GitHub）：** 存放專案程式碼的雲端中心。

• **本地倉庫（Local Repository）：** 電腦端的版本控制中心，負責儲存專案的每一版歷史紀錄。

• **工作區（Working Area）：** 開發者實際撰寫與修改程式碼的地方。

• **暫存區（Staging Area）：** 修改完程式碼後，在正式提交到本地倉庫前，需透過 `git add` 加入此區，再進行 `git commit`。

2. 同步機制：從遠端到本地

來源區分了幾種將遠端程式碼同步至本地的方式：

• **Fetch：** 將遠端的程式碼更新拉回本地倉庫，但**不會影響目前的工作區**，因此執行 Fetch 本身不會產生衝突。

• **Pull：** 等於 **Fetch + Merge**，除了拉回更新外，還會自動嘗試將其與目前工作區合併，這時就有可能產生衝突。

• **Origin vs. Upstream：** 在協作中，`origin` 通常指開發者自己的遠端倉庫，而 `upstream` 則是指原始專案（Fork 的來源）的遠端倉庫，開發者需定期從 `upstream` 同步最新的改版。

3. 合併（Merge）與 衍合（Rebase）

當需要整合不同開發者的進度時，有兩種主要策略：

• **Merge：** 將不同分支的修改合併。雖然操作直覺，但在 Git 歷史紀錄（History）上會產生較多交錯的節點，顯得較為混亂。

• **Rebase：** 意指「重新定義基準點」。它會將你目前的修改暫時移開，把本地的基準點更新到與遠端一致，再將你的修改疊加其上。

• **建議：** 使用 **Rebase** 能讓歷史紀錄保持「乾淨」，方便日後進行回溯（Rewind）或除錯。

4. 衝突解決（Conflict Resolution）

來源強調，**衝突並非 Bug（程式錯誤）**，而是因為同一個檔案的同一個地方有不同的修改，系統無法自動決定保留哪一方，需要人工審閱。

• **發生時機：** 通常在 `git push` 時，若遠端已有他人更新，系統會拒絕你的推送。這代表你的本地倉庫與遠端倉庫之間存在不一致。

• **解決流程：**

1. 先執行 `fetch` 確保拿到最新程式碼，再執行 `rebase`。

2. 開發者需打開發生命突的檔案，Git 會用「箭頭與等號（`<<<<` 、 `====` 、 `>>>>`）」標註出不同版本的差異。

3. 手動決定要保留的內容（Resolve）後，再次執行 `add` 、`commit` 並接續 `rebase --continue` 即可完成。

• **技巧：** 遇到衝突不要慌張，也不要隨意刪除程式碼，可利用編輯器的顏色區別功能，或詢問 AI 該如何下指令解決。

5. 標籤管理（Tag）

標籤（Tag）用於標記特定的重要版本（例如正式發布版）。雖然可以切換（Checkout）回標籤所在的版本觀看當時的程式碼，但該狀態下不能直接修改提交；若要從標籤版本繼續開發，必須**另開一個新分支（Branch）**。

比喻理解： Git 的運作就像是**多人共同編寫一本小說：遠端倉庫**是圖書館裡的總稿。你的**工作區**是你在家寫草稿的書桌，寫完一個段落後，先把它放進**暫存區**（信封袋），確認沒問題後鎖進你的**本地保險箱**（本地倉庫）。

當你想把新劇情寄回圖書館（`push`）時，館員發現別人已經先更新了進度，拒絕了你。這時你得先看看別人的新劇情（`fetch`），把自己的草稿重新整理，確保劇情能接在別人的進度之後（`rebase`），若發現劇情打架了，就得坐下來手動修改（解決衝突），最後才能順利寄出並讓總稿保持條理清晰。

試比較 Git 中 fetch 與 pull 的主要差異。

請解釋何謂資料庫設計中的「單一事實來源」原則。

說明 rebase 與 merge 在處理衝突時的優缺點。

Discuss 資料庫設計原則.

在開發全端應用程式時，資料庫設計是核心環節。根據來源，良好的資料庫設計不僅關乎資料的儲存，更涉及資料的**安全性、一致性、完整性與存取效率**。

以下根據來源內容，討論資料庫設計的核心原則與關鍵觀念：

1. 單一事實來源（Single Source of Truth, SSOT）

這是資訊工程中最重要的原則之一。

• **避免重複：** 資料應保存在單一位置，不應為了方便而在多處存放副本。來源提到，若同時使用 CSV 檔和資料庫儲存相同資訊，會導致資訊不同步的「悲劇」，增加維護難度。

• **透過鍵（Key）關聯：** 若要在不同資料表（Table）使用相同資訊（例如：使用者住址），應在交易表中儲存該使用者的 ID（Foreign Key），而非直接複製住址資訊，以確保資料更新時的一致性。

2. ACID 原則

來源指出，資料庫設計必須遵循 **ACID** 準則，以確保交易（Transaction）的可靠性：

• **原子性（Atomicity）：** 交易是最小的操作單位，不可分割。必須全部完成，否則就全部不執行。

• **一致性（Consistency）：** 確保資料在交易前後都符合預定義的規則。

• **隔離性（Isolation）：** 多個交易同時執行時應彼此獨立，互不干擾。

• **持久性（Durability）：** 一旦交易成功，資料就應永久保存，不會因系統故障而消失。

3. 資料結構化與正規化（Normalization）

• **Schema 設計：** 需定義資料表（Table）、欄位（Column）及其資料型態。

• **主鍵（Primary Key）：** 每個資料表都必須有一個主鍵，用來唯一識別每一筆資料（如學號或編號）。

• **正規化：** 透過將資料拆分到不同資料表並使用索引（Index）連結，消除重複資訊。例如將「客戶」、「商品」與「交易紀錄」分開存放。

4. 效能優化：索引（Indexing）

當資料量極大（如百萬等級）時，搜尋效率至關重要。

• **搜尋效率：** 若無索引，搜尋需花費線性時間 O(n)。建立索引（通常使用二元搜尋樹結構）後，可將搜尋時間降至對數時間 O(logn)。

• **權衡取捨：** 雖然索引能大幅提升讀取速度，但會增加更新（Update）與寫入的負擔，因為每次變動都需重新調整索引表。

5. 根據應用情境選擇架構

• **讀寫比（Read/Write Ratio）：** 設計前需評估讀寫頻率。例如藝術品瀏覽應用，讀取與寫入比例可能高達 **99:1**，此時適合使用輕量級的 **SQLite**；若涉及高頻率寫入的電商系統，則可能需考慮更完整的 **PostgreSQL** 或 **MySQL**。

• **OLTP vs. OLAP：**

◦ **OLTP（線上事務處理）：** 側重快速、頻繁的資料更新與交易。

◦ **OLAP（線上分析處理）：** 側重跨表分析（如分析各區消費比例），通常需透過 **ETL**（擷取、轉換、載入）過程將資料整合至資料倉庫。

類比理解： 資料庫設計就像是**設計一座自動化圖書館**。

• **SSOT** 確保每本書只有一個正本，避免不同架子上出現內容不一的副本。

• **ACID** 就像是借書程序：必須同時完成登記和領書，不能只做一半。

• **索引（Indexing）** 就像是**圖書目錄卡**：如果你想找一本書，查目錄（O(logn)）絕對比走遍整間圖書館一架一架翻（O(n)）要快得多，但缺點是每進新書，管理員都得辛苦地更新目錄卡。

為什麼 Git 指令 `git fetch` 不會造成衝突？

請比較 SQL 與 NoSQL 資料庫的適用情境。

解釋 Git 中 `primary key` 與 `foreign key` 的功用。

Discuss 後端架構實作.

在現代全端開發中，**後端架構（Backend Architecture）** 的實作是確保應用程式邏輯正確、資料安全且具備擴展性的核心。根據來源內容，後端不僅是 API 的提供者，更涉及資料庫管理、外部服務整合及安全性驗證。

以下探討後端架構實作的關鍵組成與策略：

1. 核心技術棧與執行環境

後端實作通常採用 **Node.js** 作為執行環境，這使得開發者能以 JavaScript 撰寫伺服器端程式碼。

• **Express 框架：** 用於開啟後端服務、定義路由（Routing）並連接資料庫。它能以標準化架構處理 HTTP 請求，並透過中介軟體（Middleware）擴充功能。

• **環境部署：** 為了結構清晰，建議將前後端程式碼拆分到獨立目錄（如 `frontend/` 與 `backend/`），並在伺服器入口文件（如 `server.js`）中初始化資料庫與中介軟體。

2. 路由設計與 RESTful API

後端架構的核心在於定義清晰的 API 端點，以區分不同使用者權限。

• **權限區分：** 實作上可將 API 分為「Viewer（公開）」與「Admin（受管理）」兩組。Viewer 用於讀取資料，而 Admin 則負責執行 **CRUD**（新增、讀取、更新、刪除）操作。

• **RESTful 規範：** 區分 `POST`（新增）與 `PUT`（更新）的差別，並利用參數（如 `:ID`）來處理特定資料的請求。

• **身分驗證：** 對於管理端介面，應使用 **JWT（JSON Web Token）** 進行登入管理與權限控制，確保敏感操作的安全性。

3. 資料庫設計原則

資料庫是後端的靈魂，其設計需遵循以下原則：

• **單一事實來源（Single Source of Truth, SSOT）：** 資料應保存在單一位置（如資料庫），避免重複存放副本（如 CSV 檔），以防止資訊不同步。

• **ACID 原則：** 確保交易具備原子性、一致性、隔離性與持久性。

• **性能優化：** 建立**索引（Indexing）** 將搜尋時間從 O(n) 降至 O(logn)，但需注意這會增加更新資料時的成本。

• **工具選擇：** 對於讀寫比約為 99:1 的輕量應用，建議使用 **SQLite** 以節省資源；若流量龐大或有多人併發寫入需求，則應考慮 **PostgreSQL** 或 **MySQL**。

4. 整合外部 API 與 Proxy 模式

當後端需要整合第三方服務（如 Google Drive）時，直接串接可能會遇到限制。

• **處理頻率限制：** 外部服務通常有 **Rate Limit**（如 429 錯誤）。實作上應加入一個 **Proxy（代理端點）** 或中介軟體來轉發請求，而非讓前端直接存取 API。

• **隱藏細節：** 透過 Proxy，可以為外部資源創建統一的網址（URI），隱藏底層儲存空間的真實路徑，保護系統內部結構。

5. 監控與維護機制

一個完整的後端架構需要考量「不可見」的維護工作：

• **Log 系統：** 後端程式碼運行在伺服器而非瀏覽器，發生錯誤時無法直接看 Console。因此需實作 Log 機制記錄錯誤訊息，以便診斷。

• **自動化測試：** 利用 AI 生成測試腳本（Test Scripts）進行階段性驗收，確保 API 功能符合預期。

• **監控與遷移（Migration）：** 設定監控程式追蹤資料庫用量，並在流量達標時提供遷移至更強大資料庫（如從 SQLite 轉至 PostgreSQL）的預備方案。

類比理解： 後端實作就像是一間**大型餐廳的廚房內裝與管線工程**。

• **Express 路由**是餐廳的點餐窗口，區分內用與外送（不同權限的 API）。

• **資料庫**是食材冷藏室，**SSOT** 確保同樣的食材不會散落在不同角落導致腐爛。

• **Proxy 模式**就像是在外送平台與廚房之間多設一個**出餐總監**，用來協調出餐速度，避免單一時間請求過多導致廚房癱瘓（Rate Limit）。

何謂資料庫設計中的 ACID 原則？

比較 Git 中 Fetch 與 Pull 的差異。

為何建議使用雲端代理端點連接 Google Drive？

Discuss 關聯式資料庫.

關聯式資料庫（Relational Database）是全端開發中管理結構化資料的核心技術。根據來源內容，關聯式資料庫將資料以**資料表（Table）**的格式進行儲存，其結構非常類似於 Excel 表格。

以下結合來源，從核心設計原則、技術機制與應用場景進行詳細探討：

1. 核心設計原則

關聯式資料庫的設計目標是確保資料的可靠性與一致性：

• **ACID 原則：** 這是資料庫管理的黃金標準，包含**原子性（Atomicity）**（交易不可分割）、**一致性（Consistency）**（符合規則）、**隔離性（Isolation）**（事務獨立）及**持久性（Durability）**（資料永久保存）。

• **單一事實來源（Single Source of Truth, SSOT）：** 資料應只存放在一個地方，避免重複。若在多處存放相同資料（如 CSV 與資料庫並行），會導致資訊不同步的「悲劇」。

• **正規化（Normalization）：** 透過正規化程序消除冗餘資料，確保資料庫中沒有重複資訊，若有相關聯的需求則透過「鍵（Key）」來串接。

2. 關鍵架構組件

在實作關聯式資料庫時，需要定義 **Schema（模式）**，其中包含：

• **主鍵（Primary Key）：** 用於唯一識別每一筆資料的欄位，例如學號或身分證字號。

• **外鍵（Foreign Key）：** 當不同資料表需要關聯時，透過紀錄另一張表的主鍵來連結資料，而非複製內容，這符合 SSOT 原則。

• **欄位與型態：** 每個欄位（Column）都需定義特定的資料型態（如數字、文字）。

3. 效能優化與查詢語言

• **SQL（Structured Query Language）：** 這是專門為關聯式資料庫設計的結構化查詢語言，常見指令包括 `SELECT`、`INSERT`、`UPDATE`、`DELETE`。

• **索引（Indexing）：** 為提升查詢效率，會在主鍵上建立索引（如二元搜尋樹結構）。這能將搜尋時間從線性時間 O(n) 大幅降至**對數時間** O(logn)。例如一百萬筆資料，原本需搜尋百萬次，建立索引後約 20 次即可找到。但索引會增加資料更新時的負擔，因此需在讀寫效率間取得平衡。

4. 系統選擇與適用場景

來源提到幾種常見的資料庫管理系統（DBMS）：

• **SQLite：** 極為輕量的資料庫，適合小型應用或**讀寫比極端（如 99:1）**的場景（例如個人畫展網頁，讀取頻率遠高於單人管理端的寫入）。

• **PostgreSQL / MySQL：** 較為完整且穩定的開源方案。PostgreSQL 被認為是功能最齊全、最穩定的選擇；MySQL 則在早期網頁開發中非常流行。

• **與 NoSQL 的差異：** 關聯式資料庫（SQL）適合結構嚴謹的資料；而非關聯式資料庫（NoSQL，如 MongoDB）則適合社交網路的圖形資料（Graph-based）或高頻率讀寫的非結構化文件。

5. 進階操作：合併（Join）與分析（OLAP）

當需要跨表分析資料時（例如分析北中南客戶的消費比例），開發者會使用 **Join** 操作將不同表合併，或透過 **ETL（擷取、轉換、載入）** 程序將交易型資料（OLTP）導入分析型資料庫（OLAP）。

比喻理解： 關聯式資料庫就像是一座**分類嚴謹的自動化圖書館**。每一本書（資料）都有唯一的編號（主鍵），並且只會放在一個特定的架子上（SSOT）。如果你想知道某個作者的生平，你只需查看書上的「作者 ID」，然後去作者資料區查詢（外鍵關聯）。圖書館門口有一本**精確的目錄索引**，讓你不用翻遍整座圖書館，就能在幾秒鐘內定位到書本的位置。

Git 中的 fetch 與 pull 有何不同？

寫入 Git 版本庫時，為何不會產生衝突？

為何建議在開發時保持「單一真理來源」原則？

Discuss 第三方 API 整合.

在全端應用程式開發中，**第三方 API 整合（3rd-party APIs Integration）** 是建構合理且完整網路應用的核心策略。開發者不應「自己打造輪子」，而應透過整合外部服務來擴充功能、節省開發成本並提升效率。

以下根據來源內容，探討第三方 API 整合的關鍵策略、實作流程與面臨的挑戰：

1. 整合的核心價值與策略

• **避免重複造輪子：** 現代網路應用極為複雜，利用外部成熟的 API（如 Google Maps 或 Google Drive）可以讓開發者專注於核心業務邏輯，而非底層基礎設施。

• **降低成本：** 對於儲存需求（Storage），與其自行購買昂貴伺服器，不如利用 Google Drive 等服務提供的免費或廉價容量（如 15GB 免費空間）。

• **風險管理與優先權：** 第三方 API 的整合通常具有較高的**不確定性與技術風險**。因此，在制定開發計畫時，應將其列為**第一階段（Stage 1）優先處理**，確保權限與連線正確後再進行後續開發。

2. 實作流程：以 Google 服務整合為例

整合過程通常涉及後端、雲端平台與驗證機制的協作：

• **雲端平台設定：** 開發者需在 **Google Cloud Console** 建立專案、開啟特定的 API 功能（如 Google Drive API 或 Google Maps API）並設定存取權限。

• **身分驗證（Authentication）：**

◦通常使用 **OAuth 2.0** 協議。

◦需區分「使用者帳號」與**「服務帳號（Service Account）」**。服務帳號適合用來讓伺服器自動化處理資料存取，例如代表畫家管理系統上傳畫作。

• **憑證與金鑰管理：** 需建立**憑證（Credentials）與金鑰（Keys）**。這些敏感資訊必須保留在**後端或雲端環境**中，不能讓終端使用者看到，以維護資料安全。

3. 技術挑戰與解決方案

• **速率限制（Rate Limit - 429 錯誤）：** 外部服務通常會限制每秒請求次數以防禦攻擊。若請求過於頻繁，會收到 **429 Too Many Requests** 錯誤。

◦**解決方案：** 應加入**中介軟體（Middleware）**或**代理端點（Proxy）**來統一管理與轉發請求，並加入重試（Retry）機制，而非讓前端直接存取。

• **隱藏底層細節：** 透過建立 Proxy 路由，開發者可以創造一個**統一的網址（URI）介面**（例如：`mydomain.com/image/123`），避免直接暴露第三方服務（如 Google Drive）的真實連結，同時也方便未來遷移至其他儲存服務。

• **API 規格更迭：** AI 雖然可以生成整合代碼，但有時會引用不存在或過時的 API 方法（如 `get_file_metadata`）。開發者仍需手動檢閱（Code Review）並參考最新的開發指南。

4. AI 輔助整合的作用

• **自動生成腳本與文件：** AI 擅長撰寫繁瑣的整合指南與 **測試腳本（Testing Scripts）**。

• **階段性驗收：** 在開發過程中，可以請 AI 寫一段測試程式，直接在終端機（Terminal）測試 API 是否能正確連線與回傳資料，並透過顏色標示（紅黃綠燈）快速判斷問題所在。

類比理解： 第三方 API 整合就像是**為你的餐廳外包「食材供應」與「物流服務」**。你不需要自己開農場（開發地圖系統）或組建貨車隊（開發雲端儲存），你只需要與專業供應商（如 Google）簽約並取得鑰匙（API Key）。但你必須在開店前優先測試供應商是否能順利送貨（風險優先處理），並設置一個收貨口（Proxy/Middleware）來協調進貨頻率，避免一次叫太多貨導致倉庫癱瘓（Rate Limit）。