https://youtu.be/XTuJG4i9Wmc

這份教學原始碼主要探討**單頁面應用程式（Single Page Application, SPA）**的開發邏輯，並以一個從**電子試算表（Spreadsheet）**出發的餐廳點餐系統作為實戰案例。文中深入解析了**前端路由（Client-side Routing）**的核心機制，說明網頁如何在不重新整理整頁的情況下，透過 JavaScript 動態更新局部內容，並利用路由技術維持瀏覽器的歷史紀錄功能。除了技術架構，作者特別強調在 AI 協作開發的時代，開發者應具備**系統抽象化**與**任務拆解**的能力，將 AI 視為初級工程師並給予精確的架構指引。最後，原始碼提醒開發者必須養成**程式碼重構（Code Refactoring）**的習慣，以應對隨需求增加而變得複雜的程式邏輯，避免開發陷入瓶頸。

Discuss 單頁面應用程式.

單頁面應用程式 (Single Page Application, SPA) 是現代網頁開發中極為重要且流行的一種架構做法。其核心定義在於整個 Web App 實際上只由**一個單一的 HTML 頁面**組成，使用者在操作過程中的所有互動，都是透過 JavaScript 與應用程式溝通，僅針對頁面中**部分內容進行更新**，而不會觸發整個頁面的重新刷新（Reload）。

以下根據來源內容探討 SPA 的關鍵特徵與運作機制：

1. 核心特徵與使用者體驗

• **無縫切換：** 由於 SPA 不需要重新載入整個頁面，內容的更新感覺非常自然且流暢。例如在 Google Map、Facebook、YouTube 或 Twitter 等常見服務中，都能觀察到這種頁面切換的行為。

• **瀏覽歷史紀錄 (History)：** 雖然是單一頁面，但 SPA 的網址（URL）仍會隨操作改變。透過前端路由技術，開發者可以將不同的網址路徑紀錄在瀏覽器的 history 中，讓使用者依然能使用「上一頁」或「下一頁」功能，回到先前的瀏覽狀態。

• **資料驅動 (Data-driven)：** SPA 的開發往往從資料出發。例如一個點餐系統，背後的菜單、價格等資訊可能來自於一個試算表或 CSV 檔案，前端程式碼會讀取這些資料並動態生成介面。

2. 前端路由 (Client-side Routing) 的機制

在 SPA 中，網址中的斜線（如 `/groups` 或 `/pages`）並不代表伺服器上的實際檔案目錄，而是一種**路由 (Routing)** 的概念。

• **路由對應 (Mapping)：** 前端會定義一個地圖（Map），將特定的**網址路徑**對應到特定的**組件 (Component)**。

• **佔位符與動態更新：** 以 React 為例，通常會定義一個佈局（Layout）作為框架（包含不變的 Header 或 Footer），中間則使用如 `Outlet` 的佔位符。當網址改變時，前端路由會解析該路徑，並將對應的組件插入佔位符中。

• **純前端 vs. 後端更新：** 有些更新純粹是前端 CSS 或 JavaScript 的變動（如收合選單），網址不變；有些則需要根據網址參數向後端請求資料後再進行局部更新。

3. 前後端溝通架構

SPA 的運作高度依賴前後端的分離與溝通：

• **API 服務：** 前端透過 HTTP 請求（如使用 Axios 套件）與後端的 API Service 溝通。

• **資料格式 (JSON)：** 後端從資料庫撈取資料後，通常以 **JSON (JavaScript Object Notation)** 格式回傳給前端。JSON 是一種由鍵值對（Key-Value Pair）組成的架構，前端解析後再動態畫出頁面內容。

4. 開發思維與 AI 的輔助

在開發複雜的 SPA 時（如包含購物車、送單邏輯的點餐系統），來源強調了幾點重要的軟體工程思維：

• **抽象化 (Abstraction)：** 開發者需要先建立系統的「地基」，將複雜需求抽離出核心關鍵，再逐步細化。

• **任務拆解 (Task Partitioning)：** AI（如 Cursor）雖然強大，但其 Context Window（能看到的程式碼範圍）有限。開發者必須引導 AI 進行任務拆解，否則容易產生邏輯混亂或陷入「Panic」狀態。

• **適時重構 (Refactoring)：** 當需求變得複雜時，原本簡單的架構可能產生相依性問題（Dependency issue），此時必須引導 AI 進行程式碼重構，將功能分開處理，以維持系統的可維護性。

單頁面應用程式就像是一場舞台劇： 觀眾（使用者）從頭到尾都坐在同一個劇場（HTML 頁面）裡，不需要換場地，而是透過後台工作人員快速更換佈景與道具（JavaScript 動態更新內容），讓劇情得以流暢地持續進行。

為什麼 single page application 點擊時網址不變卻能更新內容？

在前端開發中，進行 client-side routing 的主要好處為何？

若 AI 寫程式時陷入邏輯死循環，開發者應如何引導？

Discuss 前端路由機制.

前端路由機制 (Client-side Routing) 是單頁面應用程式 (SPA) 的核心技術，它改變了傳統網頁導覽的模式。在 SPA 中，前端路由負責管理網址（URL）與頁面內容之間的對應關係，讓應用程式在不重新載入整個頁面的情況下，實現流暢的畫面切換,。

以下根據來源內容詳細探討前端路由的運作機制與特點：

1. 路由的定義與本質

• **非實體目錄：** 在網址中看到的斜線（例如 `/groups` 或 `/pages`）在前端路由中並不代表伺服器上的實際檔案目錄，而是一種**路徑的分類與對應概念**,。

• **對應地圖 (Mapping)：** 前端路由本質上是一個 **Map（地圖）**，將特定的「網址路徑」與特定的「組件 (Component)」關聯起來,。當使用者訪問某個路徑時，程式會根據這個地圖決定要顯示哪一個組件。

2. 運作流程與技術細節

當使用者點擊連結或輸入網址時，前端路由會執行以下步驟：

• **網址解析：** 瀏覽器接收到網址後，前端程式會先進行解析，確認使用者要求的資源類別（例如某個 Group ID 或 User ID）。

• **局部更新與資料請求：**

◦解析完成後，前端會透過 **API Service** 發送 HTTP 請求（如使用 Axios 套件）向後端要資料,。

◦後端從資料庫撈取資料後，以 **JSON** 格式回傳,。

◦前端收到 JSON 資料後，僅針對頁面中受影響的部分進行更新，而非整頁刷新,。

• 佈局 (Layout) 與佔位符 (Outlet)： 在 React 等框架中，通常會定義一個不變的 **Layout**（包含 Header 和 Footer），並在中間設置一個名為 `Outlet` 的**佔位符（Placeholder）**,。當路由改變時，對應的頁面組件就會被插入這個佔位符中。

3. 前端路由的優勢

• 維護瀏覽歷史 (History)： 這是使用前端路由最重要的原因之一。雖然實際上還是在同一個 HTML 頁面，但透過路由更換網址，瀏覽器會將操作紀錄在 **History** 中，讓使用者能正常使用「上一頁」與「下一頁」功能,。

• 無縫的使用者體驗： 相較於傳統網頁切換時整個畫面會閃爍、重新載入（如切換不同系所網頁），前端路由能提供「無縫」的切換感，因為 JavaScript 代碼通常已預先載入，只需更新資料與元件,。

• 平行時空的概念： 透過路由，開發者可以讓不同的服務路徑具備各自的狀態，方便使用者在不同功能的頁面間切換（例如從商品列表切換回購物車），而不會遺失當前的瀏覽位置,。

4. 開發者的角色與挑戰

來源提到，雖然 AI 可以協助生成路由代碼，但開發者必須具備架構思維：

• 任務拆解 (Task Partitioning)： 開發者需要指導 AI 如何將複雜的系統拆解為不同的路由與組件,。

• 適時重構 (Refactoring)： 當系統功能增加（例如從 50 個需求增加到 100 個），原本簡單的路由架構可能會產生**相依性問題 (Dependency issue)**，此時需要引導 AI 進行程式碼重構，以維持系統邏輯的清晰,。

前端路由機制就像是計程車司機與地圖： 網址是使用者提供的「地址」，前端路由則是「司機」。司機不需要帶你換一台車（重新載入頁面），而是根據地圖（路由表）直接將車開往目的地，並更換窗外的風景（動態更新內容）。

試說明單頁式應用程式（SPA）的主要定義。

為何從 spreadsheet 出發開發 APP 具有實際應用價值？

在單頁架構中，Client-side routing 的主要優點為何？

Discuss 資料導向開發.

資料導向開發 (Data-driven Development) 是一種現代軟體開發的核心思維，其重點在於以「資料」作為應用程式的起點與核心，透過資料的結構來驅動介面的生成與功能的運作,。根據來源內容，以下探討資料導向開發的關鍵面向：

1. 以資料為起點的開發流程

在許多實際應用場景中，最基本的初衷往往來自於一份資料表（Spreadsheet），例如 Excel 或 CSV 檔案。

• **常見場景：** 學校的選課系統（背後是全校課程資料表）、餐廳的菜單（包含品項、成分、價格）、或電商的商品清單,。

• **從試算表到 App：** 開發者思考的順序是：如何從一份靜態的試算表出發，逐步建構出一個讓使用者可以互動（如點餐、搜尋、加購）的應用程式。

2. 利用 AI 產生與處理資料

來源指出，在開發初期或測試階段，開發者可以利用 AI 工具（如 Cursor）來輔助資料處理：

• **假資料生成：** 若手邊沒有實際資料，可以要求 AI 生成上百筆符合規範的假資料（例如 100 道不同的義大利麵菜單），這能讓 App 在開發初期就具備完整的展示效果,。

• **動態載入：** 前端程式會讀取這些資料（通常轉換為 JSON 格式），並根據資料內容動態畫出頁面組件,。例如，若資料中標註某道菜為「素食」，前端介面就能自動篩選並顯示。

3. 資料更新與系統維護

資料導向開發的一大優勢在於**資料與邏輯的分離**：

• **自動化更新：** 當店家需要更新資訊（如漲價、新增菜色）時，理想的情況是只需要更換底層的資料檔案，而不需要重新修改程式碼，App 就能自動反映最新的狀態。

• **降低溝通成本：** 對於不懂技術的業主來說，維護一份試算表遠比修改程式碼容易，這使得服務的維護更具彈性。

4. 資料處理的工程思維

雖然資料是核心，但將資料轉化為功能的過程仍需要嚴謹的軟體工程思維：

• **邏輯細節：** 以點餐系統為例，資料定義了菜單，但「加入購物車」、「計算總價」、「確認送單」等邏輯則需要開發者引導 AI 逐步實踐，並處理其中的小細節（如空單不可送出）,。

• **抽象化與重構：** 當資料量變大或需求變複雜時，原本簡單的結構可能會產生相依性問題（Dependency issue），開發者必須適時進行**代碼重構 (Refactoring)**，將系統架構調整得更清晰，以處理更複雜的資料互動。

• **前後端溝通：** 在實務中，資料通常存放在後端資料庫，前端透過 API 請求資料，並以 **JSON** 物件（Key-Value Pair 結構）進行傳輸，這也是資料導向開發在網路架構上的基本實踐,。

資料導向開發就像是一座自動化圖書館： 資料就是書架上的書籍。開發者並不是手寫每一本書的內容，而是設計好一套系統（App），只要管理員將新的書籍（資料）放入書架，系統就會自動將書分類、編碼並展示給讀者（使用者），而不需要為了每增加一本書就重新裝潢圖書館。

何謂單頁面應用程式 (SPA)？

前端路由 (Client Side Routing) 的主要優點為何？

當 AI 難以解決複雜程式邏輯時，建議採取什麼做法？

Discuss 抽象化思維.

在軟體工程與現代網頁開發中，**抽象化思維 (Abstraction)** 是處理複雜系統時不可或缺的核心能力。根據來源內容，抽象化不僅僅是技術手段，更是一種將複雜需求轉化為可執行邏輯的系統性思考方式。

以下深入探討抽象化思維的關鍵概念及其在開發中的應用：

1. 抽象化的本質：抓大放小

抽象化的核心在於**忽略不必要的細節，專注於系統最重要的關鍵元素**。

• 烏龜的比喻： 來源以「畫烏龜」為例，說明我們不需要畫出烏龜的每一處細節，只要透過幾個代表性的關鍵線條，就能讓人正確識別出「烏龜」的觀念。如果抽象化做得不夠精確，可能會讓人誤以為畫的是太陽或其他東西。

• 建立地基： 在開發複雜系統時，抽象化就像是**蓋房子的地基**。開發者應先在高層次 (High-level) 確定重要的關鍵點，確保基礎結構沒有大錯後，再逐步縮放 (Zoom in) 到細節處進行修改。

2. 應對複雜性與人腦極限

人類大腦無法同時記住並處理系統的所有細節，因此需要透過抽象化來管理複雜性。

• 任務拆解 (Task Partitioning)： 當系統變得極其複雜時，開發者必須具備將需求「層層拆解」的能力，先建立抽象框架，再將任務分塊處理。

• 系統性溝通： 工程師的價值在於如何「有系統地」將人類複雜的需求跟電腦溝通。無論是以前使用低階語言（如組合語言），還是現在使用高階語言或 AI，這種**將需求抽象化為邏輯**的過程本質上是一樣的。

3. 與 AI 協作中的抽象化

在 AI 工具（如 Cursor）普及的時代，抽象化思維變得更加重要，因為 AI 的運作受到 **Context Window (Token)** 的限制。

• 克服 Token 限制： AI 每次能看到的程式碼範圍有限，如果一開始就丟入過於龐大且未經抽象化的需求，AI 可能無法掌握全局結構。

• 引導 AI 的能力： 開發者必須像是一位「超級資深工程師或主管」，先建構出整體的抽象架構，引導 AI 在有限的視野內發揮最大效能。若缺乏抽象化概念，開發出的程式碼會像「違章建築」般混亂，最終導致 AI 陷入邏輯死循環（Panic 狀態）。

4. 抽象化與程式碼重構 (Refactoring)

當系統需求增加（例如從 50 個變成 100 個）時，原本簡單的架構可能會產生**相依性問題 (Dependency issue)**。此時需要透過抽象化思維來進行重構：

• 抽離與分開： 發現問題無法簡單處理時，必須將邏輯「抽離」出來或「分乾淨」，重新定義更合理的抽象層級，以維持系統的穩定性。

抽象化思維就像是地圖的比例尺： 當我們想了解城市間的交通時，會使用大比例尺地圖，略過每條巷弄的細節，只看主要幹道（核心邏輯）；唯有當我們到達特定區域時，才會切換到細節地圖。若沒有大比例尺的視野，我們將迷失在無盡的細節森林中。

何謂單頁面應用程式（SPA）？

試述前端路由與後端路由的差異。

如何利用試算表資料建構具互動性的應用程式？

Discuss 程式碼重構.

在現代軟體工程，特別是與 AI 協作開發的過程中，**程式碼重構 (Code Refactoring)** 是維持系統健康與可擴展性的關鍵活動。重構並非改變程式的外部行為，而是優化其內部結構，以因應日益複雜的需求。

以下根據來源內容探討程式碼重構的核心概念、必要性及在 AI 時代的實作方式：

1. 為什麼需要重構？

• 應對「違章建築」式開發： 在開發初期，開發者可能只跟 AI 溝通了 50 個需求，AI 會根據這些需求建立一個簡單的架構。然而，當需求增加到第 51、52 個時，原本的架構可能難以負荷，若強行加上去，程式碼就會像「違章建築」一樣變得混亂不堪。

• 解決相依性問題 (Dependency Issues)： 當系統變得複雜，不同功能之間可能會產生過度耦合的相依性，導致改動一處卻引發預料之外的 bug。此時必須透過重構將邏輯「抽離」或「分乾淨」。

• 避免 AI 陷入恐慌 (Panic)： 若程式結構混亂，AI 在協助修改時可能會陷入邏輯死循環或持續產生錯誤（即來源提到的 Panic 狀態），甚至發生「改了這裡、壞了那裡」的情況。

2. 重構的核心思維：抽象化

重構的高階目標是實現更好的**抽象化 (Abstraction)**。

• 建立穩固地基： 重構通常涉及重新定義系統的關鍵點，確保地基穩固後，再逐步細化。

• 任務拆解： 透過重構，將龐大且複雜的任務拆解成較小、更易管理的組件或模組，這有助於在 AI 有限的 **Context Window (Token)** 限制下，依然能精準地進行開發。

3. AI 時代的重構實踐

與過去相比，重構的門檻與效率已發生巨大變化：

• 從「大工程」變「隨手可得」： 以前手動重構是一個耗時數週、甚至數月的「大工程」，需要開新分支（Branch）並慢慢移植代碼。現在利用如 Cursor 等 AI 工具，重構變得非常「香（有效率）」，幾小時內就能完成以前要花一週的工作。

• 需要開發者的「技術邊界」： 雖然 AI 可以執行重構，但開發者不能「無腦」地要求 AI 重構。開發者必須具備判斷力，給予 AI 正確的指引，否則 AI 可能只會做一些表面修改，而無法真正優化架構。

4. 重構的時機

來源強調，重構應該是一個**定時的習慣**。

• 及時處理： 不要等到程式碼已經「病入膏肓」或完全無法維護時才要求 AI 重構，那時往往已經來不及了。

• 預見需求： 當發現新需求與現有架構產生衝突時，就是進行重構的最佳時機。

程式碼重構就像是「整理房間」： 當你不斷買新東西（加新功能）進房間，若不隨時重新規劃收納空間（重構架構），房間最終會亂到連路都走不動（程式當機或 bug 叢生）。及時的整理能讓你以後放更多東西時，依然井然有序。

何謂單頁式應用程式（SPA）？

前端路由與後端路由的主要差異為何？

為何建議從 Spreadsheet 出發來設計應用程式？