

[https://www.youtube.com/watch?v=c-jrCpxyG34&list=PLIAzIZzCjtLJQzoK-1n6R\\_KVTXm4tHZqP&index=14](https://www.youtube.com/watch?v=c-jrCpxyG34&list=PLIAzIZzCjtLJQzoK-1n6R_KVTXm4tHZqP&index=14)

[https://youtu.be/ExiKZldmkX4?list=PLIAzIZzCjtLJQzoK-1n6R\\_KVTXm4tHZqP](https://youtu.be/ExiKZldmkX4?list=PLIAzIZzCjtLJQzoK-1n6R_KVTXm4tHZqP)

這份教學原始碼主要探討**全端應用程式**的建構流程與核心技術觀念，並透過實作一個具備資料庫功能的 **To-Do List** 作為示範。課程詳細說明了 **npm** 與 **yarn** 等套件管理工具的差異，並強調 **package.json** 與環境變數檔案 (**.env**) 在版本控制與隱私安全中的重要性。教學重點聚焦於 **JavaScript** 的**非同步處理機制**，包含 **Blocking** 與 **Non-blocking** 的區別，以及如何運用 **Async/Await** 語法簡化複雜的程式執行順序。此外，作者介紹了現代 AI 開發工具 **Cursor** 的 **Plan Mode**，展示如何透過預先規劃步驟實現全端應用的 **One-shot Coding**，並利用 **ngrok** 快速完成本地開發環境的對外部署。

這份教學資源深入淺出地解析了**全端應用程式**的架構，核心圍繞著如何運用 **PostgreSQL** 與 **Prisma ORM** 建立穩定且具備型別安全的後端服務。課程重點比較了**關聯式與非關聯式資料庫**的優劣，指出關聯式結構在數據統計上的優勢，並強調透過 **Primary Key** 與 **Foreign Key** 優化查詢與資料關聯效率。除此之外，內容特別介紹了 **ngrok 反向代理工具**，說明開發者如何將本地環境快速對外展示以進行測試。最後，講者分享了**敏捷式開發**的思維，鼓勵學生在 AI 輔助開發的時代應培養**批判性思考**與具備想法的能力，並透過分階段切入複雜問題，以確保持續交付可執行的開發成果。

Discuss 全端應用開發.

全端應用開發 (Full-stack Application Development) 是一個涵蓋**前端介面**、**後端邏輯**以及**資料庫管理**的綜合過程。根據來源，現代全端開發不僅僅是編寫程式碼，還涉及工具鏈管理、非同步處理機制以及新興的 AI 協作開發流程。

以下是根據來源整理的全端開發關鍵面向：

### 1. 現代化 AI 協作開發流程

現在的全端開發可以利用 AI 工具 (如 Cursor) 的 **PL mode (Plan mode, 計畫模式)** 來大幅提升效率。

- **計畫導向**：開發者與 AI 討論需求後，AI 會生成一個 `plan.md` 檔案，將開發步驟拆解成多個步驟 (如：初始化後端、建立資料庫、撰寫 API、前端重組等)。
- **自動化執行**：AI 可以按步驟執行並在每一步自動打上 Git tag，甚至進行測試，實現所謂的「One-shot coding」。

### 2. 後端核心技術與資料庫管理

後端開發的核心在於如何穩定地存取資料並提供接口。

- **資料庫存取**：來源提到使用 **PostgreSQL** 等關聯式資料庫，並透過 **Prisma** 這種 **ORM (Object-Relational Mapping)** 工具，讓開發者能以較高階的方法操作資料庫，而非撰寫低階的 SQL 指令。
- **資料驗證 (Validation)**：在資料庫層級定義 Schema (架構) 非常重要，這能確保使用者輸入的資料符合規範 (例如年齡必須是數字)，避免錯誤資料進入系統。
- **環境變數管理**：敏感資訊 (如資料庫密碼、API 金鑰) 應放在 **.env** 檔案中，且絕不應提交 (push) 到 Git 雲端。開發者通常會提供 `.env.example` 作為範本，讓其他協作者參考並填入自己的金鑰。

### 3. 套件管理與版本控制

全端專案通常依賴大量的第三方套件，管理這些工具是開發者的基本功。

- **工具選擇**：**npm** 是官方工具，而 **yarn**（由 Facebook 開發）則因其更快速、安全的特性受到青睞。**npx** 則用於執行一次性的指令，避免污染全域環境。
- **版本一致性**：`package.json` 負責定義套件版本，而 **yarn.lock** 或 `package-lock.json` 則確保所有開發者安裝的套件版本完全一致，避免「在我的電腦可以跑，你的不行」的問題。

### 4. 非同步處理機制 (Asynchronous)

這是全端開發中最具挑戰性的觀念之一。

- **Blocking vs. Non-blocking**：傳統的執行方式是 Blocking（阻塞），即 A 函式必須等 B 函式執行完才能繼續。但在處理網路請求或資料庫 IO 時，為了不讓介面卡住，會採用 **Non-blocking（非阻塞）** 方式。
- **Async/Await**：JavaScript 本質是單執行緒且同步的，但透過 `async` 和 `await` 語法糖，開發者可以更簡潔地處理非同步溝通，避免陷入早期的「回呼地獄（Callback Hell）」。

### 5. 通訊協定與佈署

- **HTTP 與 WebSockets**：HTTP 是被動式溝通，前端主動請求，後端才回傳。若要開發如聊天室般的即時應用，則需要支援雙向溝通的 **WebSockets** 技術。
- **反向代理與測試**：在開發初期，可以使用如 **ngrok** 的工具建立通道，將地端的服務對外開放成公開網址，方便進行跨裝置或多人的即時測試。

---

**比喻理解**：全端開發就像經營一家餐廳。**前端**是精美的菜單與外場服務，**後端**是忙碌的廚房，**資料庫**則是儲藏食材的倉庫。**非同步溝通**就像是服務生送單到廚房後，不需要站在那裡等菜煮好，而是可以繼續為其他客人點餐（Non-blocking），直到廚師敲鐘（Callback）通知菜好了，服務生再回來送餐，這樣餐廳的運作才會順暢。

什麼是 npm、Yarn 與 npx 的主要區別？

為什麼在開發 Node.js 時建議使用 nodemon？

請解釋 JavaScript 中 blocking 與 non-blocking 的差異。

Discuss 套件管理工具.

在全端開發的領域中，**套件管理工具（Package Management Tools）** 是管理專案依賴項、版本控制以及執行指令的核心。在 Node.js 生態系中，最常見的工具包括 **npm**、**yarn** 以及 **npx**。

以下是根據來源整理的詳細討論：

#### 1. 核心工具對比：npm 與 yarn

- **npm (Node Package Manager)**：這是 **Node.js 官方推出**的套件管理工具，主要負責套件的安裝、更新、移除以及管理模組間的依賴關係（dependency management）。

- **yarn**：由 **Facebook** 開發，起因是早期的 npm 在效能與安全性上表現不夠理想。yarn 被認為**更快速、更可靠**，且指令較為簡潔。來源建議，除非習慣官方工具，否則通常較偏好使用 yarn。

## 2. 執行工具：npx

**npx** 與上述兩者不同，它不算是管理工具，而是一個**執行工具**。

- **避免污染**：npx 允許開發者在不進行「全域安裝」的情況下執行指令，這能避免污染電腦的系統環境。
- **一次性執行**：對於版本更新極快或僅需執行一次的工具，npx 能確保開發者使用的是最新或正確的版本，且僅在 local 環境運行。

## 3. 關鍵配置檔案

套件管理工具會在專案中產生幾個至關重要的檔案，用於確保開發環境的一致性：

- **package.json**：這是專案的控制中心，紀錄了專案名稱、版本，以及最重要的 **scripts**（執行腳本）與 **dependencies**（套件依賴）。例如，透過 `npm run start` 執行的指令就是在這裏定義的。

- **鎖定檔** (**yarn.lock** / **package-lock.json**)：

- yarn 使用 `yarn.lock`，npm 使用 `package-lock.json`。

- **重要性**：這些檔案會**鎖定套件的確切版本**，確保所有開發者在不同電腦上安裝的套件版本完全一致，避免出現「在我的電腦可以跑，你的不行」的問題。因此，這些鎖定檔**必須提交 (push) 到 Git**。

## 4. 版本控制邏輯 (Semantic Versioning)

在 `package.json` 中，套件版本號前常帶有特殊符號：

- **^ (Caret)**：相容於主版本號 (Major) 相同的所有次版本，例如 `^1.0.0` 表示支援 1.x.x。
- **~ (Tilde)**：僅相容於修補版本 (Patch)，例如 `~1.0.0`。
- **固定版本**：若不寫符號則鎖定特定版本。來源**不建議固定版本**，因為這可能導致與其他協作者的環境不相容。

## 5. 隱私與環境變數管理 (.env)

專案中常涉及敏感資訊（如資料庫密碼、API 金鑰），這些不應直接寫在程式碼或 `package.json` 中。

- **.env 檔案**：用於存放敏感金鑰。**絕不能將此檔案 push 到雲端**。
- **.env.example**：開發者應提供此範本檔（僅含變數名稱不含真實數值），讓其他協作者參考並填入自己的金鑰。

---

**比喻理解**：套件管理就像是**工地的工具箱管理**。`package.json` 是**清單**，記錄了需要哪些扳手或螺絲起子；`yarn.lock` 則是**精確的規格書**，確保每個人拿到的扳手尺寸都精確到毫米。而 `npx` 就像是**租借服務**，當你需要一個不常用的特殊大型機具時，不需要買下來佔空間，租來用完就歸還，保持工地的整潔。

什麼是 JavaScript 中的 Callback Hell？

Promise 機制如何解決非同步呼叫的問題？

試說明 package.json 與 yarn.lock 的差異。

Discuss 非同步程式設計。

**非同步程式設計 (Asynchronous Programming)** 是全端開發中至關重要的觀念，特別是在處理網路請求、資料庫存取等較慢的 I/O 操作時，能確保應用程式（尤其是前端介面）保持順暢而不卡頓。

以下根據來源討論非同步程式設計的核心概念與技術演進：

### 1. 核心觀念：阻塞與非阻塞 (Blocking vs. Non-blocking)

這兩個概念經常與同步/非同步混用，但在技術細節上略有不同：

- **阻塞 (Blocking)**：當函式 A 呼叫函式 B 時，A 必須等到 B 執行完畢後才能繼續執行剩下的程式碼。在 JavaScript 中，預設的陳述句 (Statements) 多是阻塞且同步的。
- **非阻塞 (Non-blocking)**：當函式執行到一半需要進行耗時操作（如向後端請求資料）時，它不會停下來等待，而是繼續執行後續動作，直到操作完成後再回來處理結果。
- **區分**：來源提到，**同步/非同步**通常描述兩個以上模組間的溝通（如 Host 與 I/O 裝置），而**阻塞/非阻塞**則常用於描述同一個模組內程式碼的執行行為。

### 2. JavaScript 的執行機制

JavaScript 本質上是**單執行緒 (Single-threaded)**的，這意味著它一次只能做一件事。為了在單執行緒下實現非同步溝通，JavaScript 引擎（如 Node.js）利用了 **Event Loop (事件循環)** 與 **Queue (佇列)** 的機制。這讓開發者可以發出非同步呼叫，引擎會記錄這些呼叫並在完成後觸發對應的處理程序，而不會擋住主執行緒。

### 3. 技術工具的演進

為了管理非同步流程，開發者使用的工具經歷了幾個階段：

- **回呼函式 (Callback)**：最早的方法，但在處理多層嵌套的非同步邏輯時，會產生代碼難以閱讀與維護的「**回呼地獄 (Callback Hell)**」。
- **Promise (ES6)**：解決了回呼地獄。Promise 是一個物件，承諾會回傳執行結果（成功或失敗），並允許使用 `.then()` 進行鏈式呼叫，讓程式碼結構更清晰。
- **Async/Await (ES8)**：這是 Promise 的**語法糖 (Syntax Sugar)**。透過在函式前加上 `async`，並在非同步動作前加上 `await`，開發者可以用類似「同步寫法」的方式來撰寫非同步程式碼，大幅減輕開發負擔。

### 4. 順序管理的重要性

在非同步環境中，多個任務是「同時」發出的。如果不使用 `await` 來明確指定等待，任務完成的順序是無法保證的。例如，若「付款 (payment)」動作依賴於「取得訂單 (order)」，就必須使用 `await` 確保訂單先完成，否則程式會因順序錯誤而崩潰。來源中舉例，若在非同步函式中使用 `await` 延遲三秒，外部的執行順序會因為非同步特性而與傳統同步執行產生倒置的現象。

**比喻理解：**非同步程式設計就像一場餐廳約會。**同步 (Blocking)** 的約會是你點餐後，服務生和廚師去煮菜，你必須坐在座位上發呆、不能跟對象聊天，直到菜端上來才能開口；而**非同步 (Non-blocking)** 則是單子送出後，在廚房備餐的期間，你可以繼續跟對方談笑風生，直到菜煮好了被端上桌（觸發 Callback），這才打破原本的聊天流程來用餐。

什麼是 package.json 檔案中的 script 欄位用途？

請比較 npm、yarn 與 npx 之間的功能差異。

在專案開發中，使用 .env 檔案的主要目的是什麼？

Discuss 資料庫架構設定.

在全端應用開發中，**資料庫架構設定 (Database Schema Setup)** 是確保資料一致性與系統穩定性的核心步驟。根據來源，現代開發流程傾向於使用高階工具來簡化低階的資料庫操作，並在架構層級強化資料驗證。

以下是資料庫架構設定的關鍵要素：

#### 1. 核心工具與技術選擇

- **關聯式資料庫 (Relational DB)：**來源以 PostgreSQL 為例，展示如何在本地端機器建立資料庫伺服器。
- **ORM 工具 (Object-Relational Mapping)：**由於直接編寫 SQL 指令較為低階且繁瑣，開發者通常會使用 Prisma 這樣的應用層套件。Prisma 作為 ORM 工具，讓開發者能以**更高階、更具可讀性**的方法來存取和操作資料庫，而不必處理複雜的 SQL 語法。

#### 2. 資料庫架構 (Schema) 與驗證 (Validation)

在定義資料庫時，設定明確的 **Schema (架構)** 至關重要，這能作為資料的第一道防線：

- **型別驗證：**在架構設定中規範欄位的資料型別（例如：「年紀」欄位必須是數字）。
- **早期攔截錯誤：**透過架構層級的驗證，系統可以在使用者填寫表單的「第一時間」就檢查輸入是否有誤，避免錯誤的資料進入後端甚至存入資料庫，減少後續資料搜尋或處理時的除錯困難。

#### 3. 開發流程中的資料庫建置

利用 AI 工具（如 Cursor）的計畫模式（Plan mode），資料庫的建置通常被拆解為以下步驟：

- **初始化：**在後端專案初始化後，接著進行資料庫的建立。
- **同步設定：**利用 Prisma 進行資料庫的具體設定（如定義 Model 與執行 Migration），將定義好的 Schema 同步至資料庫伺服器中。

#### 4. 安全性管理與環境變數

資料庫的存取權限（如密碼、連線字串）屬於敏感資訊：

- **.env 檔案：**所有的資料庫金鑰與路徑應存放在 .env 檔案中，**絕對不可提交 (push) 到 Git 等版本控制系統**，以防外洩。

- **範本提供**：開發者應提供 `.env.example` 檔案，列出必要的變數名稱（不含真實數值），讓其他開發者或佈署環境能根據範本填入各自的資料庫資訊。
- 

**比喻理解**：資料庫架構設定就像是在蓋圖書館。**PostgreSQL** 是存放書本的建築物，**Schema** 則是書架上的標籤和索引規則（規定哪一區只能放科學書，哪一區放小說）。而 **Prisma** 就像是一位專業的圖書管理員，你只需要告訴他「我要找某人的書」，他就會幫你處理好所有翻找架位的瑣事，讓你不需要親自去記住每一本書在底層倉庫的精確經緯度。

為什麼建議將 `package.json` 提交至 Git 進行版本管理？

在 JavaScript 中，`async` 與 `await` 的主要用途為何？

使用 `.env` 檔案管理金鑰時，如何確保隱私資訊不外洩？

Discuss AI 輔助開發.

AI 輔助開發（AI-assisted coding）在近年經歷了巨大的技術躍遷，從早期的簡單程式碼補全，演進到如今能夠處理跨檔案修改、自主決策甚至規劃完整開發流程的階段。

以下根據來源討論 AI 輔助開發的關鍵技術、模式與流程：

#### 1. 開發工具的演進：從 Copilot 到 Cursor

AI 編碼工具的發展可分為幾個重要里程碑：

- **早期階段**：以 Microsoft Copilot 為代表，主要提供簡單的程式碼提示與單一檔案處理。
- **多檔案編輯 (Composer)**：來源提到 **Composer**（由講者兒子開發，後被 Cursor 採用）是首個實現**跨多檔案一次性修改**的工具，讓 AI 能夠理解整個專案的結構並進行聯動式的代碼變更。
- **整合開發環境 (IDE)**：像 Cursor 這樣的工具因優化了 AI 與編輯器的整合，逐漸成為開發者的首選。

#### 2. 代理模式與電腦使用 (Agentic Mode & Computer Use)

現代 AI 工具已進入 **Agentic mode (代理模式)**，這意味著 AI 不僅是「寫程式」，還具備以下能力：

- **自主判斷**：AI 會根據寫出來的內容判斷下一步該做什麼，而不僅僅是單向接收指令。
- **電腦使用 (Computer Use)**：透過開放權限，AI 可以操作滑鼠、查看瀏覽器、讀取 Console log，甚至主動插入除錯代碼 (Debug code) 來修復錯誤。
- **流程自動化**：在 IC 設計 (EDA) 等複雜產業中，AI 被用來自動化繁瑣的工具流，減少開發者的無謂等待。

#### 3. 計畫模式 (PL mode / Plan mode)

這是目前全端開發中最具革命性的功能之一，讓 **「One-shot coding (一次性完成編碼)」** 成為可能。

- **制定計畫**：當開發者提出複雜需求（如將前端 App 改成全端）時，AI 會先進入 PL mode，透過與開發者討論，生成一個 `plan.md` 檔案。

- **步驟拆解**：該計畫會將任務拆解成多個邏輯步驟（例如：1. 初始化後端、2. 設定 Prisma 資料庫、3. 撰寫 API、4. 重組前端 SRC 目錄等）。

- **受控執行**：開發者可以檢視並修改這份計畫，確認無誤後再讓 AI 執行。AI 在執行每一步驟後，甚至會自動進行測試並打上 **Git tag**，方便開發者進行 Code review 或在出錯時回溯。

#### 4. 全端開發中的具體應用

AI 輔助開發極大地減輕了全端專案的建置負擔，讓教學或開發能覆蓋更多技術細節：

- **資料庫設定**：AI 可以自動化處理 PostgreSQL 與 Prisma 的整合設定。

- **通訊與介面**：AI 能輔助建立 RESTful API，並使用套件處理繁瑣的 HTTP Header 設定。

- **部署與代理**：AI 甚至能幫忙撰寫啟動腳本（如 `start.sh`），並整合 **ngrok** 等反向代理工具，在開發初期快速建立公開測試網址。

#### 5. 開發者的角色與限制

儘管 AI 能力強大，來源仍提醒開發者應保持謹慎：

- **理解極限**：開發者應了解 AI 的能力上限（Capacity limit），不要一次要求過於激進或極端複雜的任務，否則容易產生錯誤。

- **主導計畫**：AI 提供的 `plan.md` 需要由人來核實，確保架構符合預期，避免 AI 做出「愚蠢的決定」。

---

**比喻理解**：AI 輔助開發就像是從「拿著地圖自己開車」進化到了「僱用一名高級特助」。早期的 AI 只是幫你導航（Copilot），現在的 AI（如 Cursor 的 PL mode）則是先跟你開會討論出差計畫（`plan.md`），確認後由他負責訂票、聯絡旅館並處理突發狀況（Agentic mode），而你只需要負責最後的審核與決策。

試比較 npm 與 Yarn 的開發背景及優點。

說明 `package.json` 與 `yarn.lock` 檔案的用途。

解釋 JavaScript 中 blocking 與 non-blocking 的差異。

Discuss 關聯式資料庫。

關聯式資料庫（Relational Database, RDB）是現代應用開發中儲存與管理資料的主流技術。根據來源，其核心概念是將資料組織成一張或多張大型的**表格（Table）**，並透過結構化的方式進行管理。

以下是根據來源整理關於關聯式資料庫的詳細討論：

##### 1. 核心結構與元素

- **表格組成**：每個資料庫包含多張表，表由**列（Row）**與**行（Column）**組成。

- **列（Row）**：代表每一筆獨立的資（Entry）。

- **行（Column）**：代表資料的屬性（Attribute）。

- **關鍵機制：**

- **Primary Key (主鍵, PK)：** 作為每筆資料的唯一索引。為了避免從頭掃描硬碟造成的效能浪費，系統會利用資料結構（如 B-Tree 或 Hash）建立 PK，讓搜尋時間從線性減少到  $O(\log n)$  甚至常數時間。

- **Foreign Key (外鍵, FK)：** 當有多個表格時，FK 用於指向另一個表格的特定項目，藉此將多張表的資料連結起來，**避免重複存儲相同資料**。

## 2. 優缺點與適用場景

- **優點：** 資料結構清晰、一目瞭然，且非常適合進行**統計、加總與分析**。

- **缺點：** 在硬碟中通常採連續儲存，因此**頻繁刪除資料**會導致效率低下（因為剩餘資料需要移動以填補空間）。

- **選型建議：**

- **SQL (關聯式)：** 適合較穩定、變動少且需要嚴謹統計的資料，例如**會員資料**。

- **NoSQL (非關聯式)：** 適合互動頻繁、結構多變的資料，例如**社群貼文**或**互動訊息**。

## 3. 開發工具與實作

- **資料庫軟體：**

- **PostgreSQL：** 來源強烈推薦的開源選擇，經過數十年發展已非常成熟，足以支援商用需求。

- **SQLite：** 一種輕量級的資料庫，安裝與建置極快，非常適合教學或初期的地端開發。

- **ORM (Object-Relational Mapping)：**

- 直接撰寫低階的 SQL 指令較為繁瑣，因此開發者常使用如 **Prisma** 這樣的應用層套件。

- Prisma 讓開發者能以**物件導向**的高階寫法來存取資料庫，並提供型別安全（Type-safe）的保障。

- **Schema 與驗證：** **Schema (架構)** 是資料庫的設計圖，定義了表格、欄位型別及其關係。透過 Schema 定義（如使用 Zod 等工具），可以在資料進入資料庫前就進行驗證（如：年紀欄位必須是數字），避免無效資料導致後續維護困難。

---

**比喻理解：** 關聯式資料庫就像是一本管理完善的 **Excel 活頁簿**。每一張**工作表**（Table）都處理特定的主題（如：客戶清單、訂單明細），而每張表的第一欄（Primary Key）就像是身分證號碼，確保我們能快速找到對的人。如果你想知道某個客戶買了什麼，你不需要在客戶表裡重複寫下產品名稱，只需要寫下訂單編號（Foreign Key），就能像翻書籤一樣跳到訂單表找到完整資訊。

為什麼建議開發階段使用 nodemon 而非 node？

簡述 Prisma 在應用程式層扮演的角色與功能。

何謂「反向代理」以及 ngrok 的主要用途？

Discuss Prisma ORM.



**Prisma ORM** 是現代全端開發中非常受歡迎的**應用層 (Application Layer)** 資料庫存取工具，特別是在 TypeScript 生態系中被廣泛採用。它的核心定位是 **ORM (Object-Relational Mapping)**，旨在讓開發者能以**物件導向的高階語法**操作資料庫，而不需要撰寫低階且繁瑣的 SQL 指令。

以下是根據來源整理的 Prisma ORM 關鍵特性與運作流程：

### 1. 以 Schema 為核心的設計

Prisma 的運作基礎是一個名為 **schema.prisma** 的檔案，這被視為資料庫的**設計圖 (Blueprint)**。

- **定義模型**：開發者在此檔案中定義資料模型 (Data Model)，包含表格名稱、欄位、資料型別 (如 String, Int) 以及預設值。
- **關聯管理**：透過 Schema 可以明確定義表格間的關係，例如透過 **Primary Key (PK, 主鍵)** 與 **Foreign Key (FK, 外鍵)** 建立連結。
- **同步機制**：當 Schema 定義完成後，開發者執行 **prisma migrate** 指令，Prisma 就會自動在資料庫伺服器中建立對應的表格結構，並確保程式碼與資料庫同步。

### 2. 強型別與型別安全 (Type-safe)

Prisma 的一大優勢在於它能與 TypeScript 完美結合。

- **自動生成 Client**：Prisma 會根據 Schema 自動產生專屬的 **Prisma Client**，這是一個符合型別規範的程式碼介面。
- **預防錯誤**：由於具有 **Type-safe (型別安全)** 特性，如果開發者試圖傳入不符合定義的資料 (例如要求整數卻傳入物件)，系統會拒絕接收並報錯，這能在開發階段就攔截潛在的錯誤。

### 3. 簡化的 CRUD 操作

在後端的 **Controller (控制器)** 邏輯中，Prisma 讓資料的操作變得非常直觀。

- **非同步處理**：Prisma 的操作本質上都是 **Promise 物件**，因此開發者通常會搭配 `async` 與 `await` 語法來進行非同步溝通。
- **操作方法**：開發者可以呼叫如 `findMany` (查詢多筆)、`create` (新增)、`update` (更新) 或 `delete` (刪除) 等內建函式來執行 **CRUD** 動作。
- **進階查詢**：支援在查詢時加入條件過濾 (`where`) 或排序 (`orderBy`) 等參數。

### 4. 開發輔助工具

- **Prisma Studio**：來源提到 Prisma 內建了一個**視覺化視窗工具**，讓開發者可以直接透過瀏覽器查看並操作目前資料庫中的資料狀態，而不需要使用複雜的第三方資料庫管理軟體。
  - **優雅退出機制**：在後端伺服器終止 (如按下 Ctrl+C) 時，良好的開發習慣是呼叫 `prisma.$disconnect()`，確保資料庫連線被正確關閉，避免資料受損。
-

**比喻理解：** Prisma ORM 就像是資料庫與程式碼之間的**專業翻譯官兼品質檢測員**。原本你必須用資料庫聽得懂的土語（低階 SQL）溝通，但現在你只要寫下標準的說明書（Schema），翻譯官（Prisma Client）就會幫你將程式碼翻譯成指令，並且在發現你給的資料不對（型別錯誤）時立刻提醒你，確保資料庫這座倉庫裡存放的東西永遠井然有序。

什麼是 blocking 與 non-blocking 觀念？

Prisma ORM 在應用程式架構中扮演什麼角色？

開發者為何在專案中使用 ngrok 工具？

Discuss 全端架構整合。

**全端架構整合（Full-stack Architecture Integration）**是指將前端介面、後端邏輯以及資料庫系統緊密結合，使之成為一個完整運作應用程式的過程。根據來源，現代的全端開發不再只是手寫程式碼，更強調透過工具鏈（Toolchain）與 AI 輔助來達成高效的整合與自動化。

以下是全端架構整合的關鍵組成部分與整合流程：

### 1. 前後端與資料庫的串接邏輯

在一個完整的全端架構中，資料的流動遵循特定路徑：

- **前端（Front-end）：** 使用者操作的介面，透過 **Axios** 等套件發送 HTTP 請求（RESTful API）與後端通訊。
- **後端（Back-end）：** 接收前端請求，處理商業邏輯。來源建議使用 **Router** 來定義路徑，並由 **Controller** 負責具體的邏輯實踐。
- **資料庫（Database）：** 儲存持久性資料。為了提升開發效率，通常會透過 **Prisma (ORM)** 這一層來進行高階存取，避免直接編寫低階 SQL。

### 2. 資料驗證與型別安全

整合過程中的一個核心挑戰是確保資料在各層之間傳遞時的正確性：

- **Schema 定義：** 在資料庫層級定義 **Schema**，規範欄位型別（如年紀必須是數字），這能作為資料的第一道防線，避免無效資料進入系統。
- **型別安全（Type-safe）：** 使用 Prisma 等工具可以確保程式碼符合型別規範，若傳入錯誤型別，系統會直接拒絕，減少除錯成本。

### 3. 整合自動化與指令管理

為了確保全端應用的前後端能順利啟動並保持同步，開發者通常會建立自動化腳本：

- **啟動腳本（\*\*start.sh\*\*）：** 整合過程涉及多個步驟，包含檢查環境、安裝套件、執行資料庫遷移（Migration）以及啟動前後端服務。使用指令如 `&` 可以讓服務在背景執行，避免阻塞終端機。
- **環境變數（\*\*.env\*\*）：** 整合過程中涉及的金鑰或密碼應存放於 `.env` 中，且**嚴禁提交至 Git**，以確保安全性。

#### 4. 現代 AI 輔助整合流程 (PL Mode)

來源強調了使用 AI 工具 (如 Cursor) 的 **PL mode (計畫模式)** 來達成「One-shot coding」：

- **制定計畫**：AI 會根據需求生成 `plan.md`，將整合任務拆解為：初始化後端、建立資料庫、撰寫 API、重組前端目錄、前後端串接等步驟。
- **分步執行與回溯**：AI 在執行每一步後會自動進行測試並打上 **Git tag**，方便開發者進行 Code review 或在整合出錯時快速回溯。

#### 5. 溝通機制與即時性

整合時需考慮通訊協定的特性：

- **HTTP**：屬於被動式溝通，前端主動請求，後端才回應。這在多使用者同時操作時，無法即時更新其他人的畫面。
- **WebSockets**：若要整合如聊天室般的雙向即時功能，則需採用 WebSockets 技術，讓後端能主動通知前端資料的變動。

#### 6. 開發初期的公開測試 (Reverse Proxy)

在整合開發階段，為了讓外部裝置 (如手機) 或協作者能存取地端的服務，常使用 **ngrok** 建立通道。這是一種**反向代理 (Reverse Proxy)** 機制，會產生一個臨時的公開網址 (如 `-fe.app`)，將外部流量導入本地的特定連接埠 (如 3000 port)。

---

**比喻理解**：全端架構整合就像是在蓋一棟**智慧住宅**。**前端**是牆上的觸控面板 (介面)，**後端**是埋在牆裡的電線與中央處理器 (邏輯)，**資料庫**則是後院的蓄水池 (儲存)。**啟動腳本**就像是總開關，一按下去會自動檢查水電並啟動系統。而 **ngrok** 就像是幫這棟還沒對外開放的私人住宅暫時申請一個臨時門牌，讓朋友可以透過地圖導航過來參觀測試。

請比較 npm 與 Yarn 的開發背景與優點。

何謂 blocking 與 non-blocking 的執行觀念？

簡述 Prisma ORM 在全端應用中的角色。

Discuss 局部開發工具。

在全端應用開發中，**局部開發工具 (Local Development Tools)** 指的是開發者在個人電腦 (地端) 環境中使用的各種軟體與配置，用以模擬、建置、測試及管理應用程式。

以下根據來源討論局部開發工具的關鍵面向：

##### 1. 資料庫管理工具

在開發初期，開發者通常會在自己的機器上建立資料庫伺服器進行測試。

- **PostgreSQL**：來源提到可以在地端安裝 PostgreSQL Server (建議使用 v14 穩定版) 作為關聯式資料庫的後端。

- **SQLite**：這是一種輕量級的資料庫，其優點是安裝與建置極快，非常適合教學或初期的地端開發需求。
- **Prisma Studio**：這是一個視覺化的局部工具，讓開發者能直接透過瀏覽器查看目前地端資料庫的內容與狀態，而不需要下複雜的指令。

## 2. 執行與監控工具

為了提升開發效率，地端環境需要特定的工具來管理套件與執行程式：

- **npx**：來源指出 `npx` 是一個執行工具，它允許開發者執行一次性指令並將套件裝在「局部 (local)」而非全域環境中，這能有效**避免污染系統環境**，並確保使用正確的版本。
- **Nodemon**：這是在開發階段非常實用的局部工具。它具備 **Hot Reload (熱重載)** 功能，只要開發者在地端修改並存檔，Nodemon 會自動偵測變更並重啟後端服務，省去手動重啟的麻煩。

## 3. 反向代理與公開測試 (ngrok)

當開發者在地端完成應用後，若想讓外部裝置（如他人的手機）存取自己電腦上的服務，會使用 **ngrok**：

- **建立通道**：`ngrok` 是一個反向代理工具，能將地端的連接埠（如 `localhost:3000`）映射到一個**臨時的公開網址**。
- **開發展示**：這讓開發者不需正式佈署到雲端，就能在開發初期進行跨裝置測試或向他人展示成果。

## 4. AI 協作 IDE (Cursor)

現代局部開發環境常整合強大的 AI 工具，如 **Cursor**：

- **PL mode (計畫模式)**：開發者可以在局部環境中與 AI 討論開發計畫，AI 會生成 `plan.md` 並分步驟執行，甚至在每步自動打上 **Git tag**，實現「One-shot coding」的開發流程。
- **Composer**：這是 Cursor 的核心功能之一，支持跨多個局部檔案進行一次性的程式碼修改。

## 5. 隱私與配置管理 (.env)

局部開發中涉及的敏感資訊（如金鑰、密碼）會存放在 **.env** 檔案中。

- **環境隔離**：這些變數會被讀取到局部系統的環境變數（`process.env`）中，確保開發者不需要將真實密碼寫在程式碼裡，從而避免將敏感資訊提交到公開的雲端空間。

---

**比喻理解**：局部開發工具就像是**建築師的個人工作室**。**PostgreSQL** 是工作室裡的小型模型台；**Nodemon** 就像是一個自動助理，每當你改動草圖，它就立刻幫你重新組裝模型；而 **ngrok** 則是在工作室牆上開了一扇臨時的窗戶，讓路過的人（外部使用者）可以直接看到你桌上的半成品並給予回饋，而不需要等到整棟房子蓋好。

試比較 `npm` 與 `yarn` 的開發背景與優點。

請解釋環境變數檔 `.env` 在隱私管理中的作用。

什麼是 `ngrok`？它如何協助本地端開發測試？

Discuss 敏捷開發思維.

**敏捷開發思維 (Agile Development Thinking)** 在現代軟體工程中至關重要，特別是在 AI 工具大幅提升編碼效率的背景下。根據來源，敏捷開發的核心在於**碎片化任務、持續驗證以及從「苦工」轉向「思考」**的轉變。

以下是根據來源整理的敏捷開發核心面向：

### 1. 避免「大爆炸式」開發 (Big Bang Development)

敏捷開發最忌諱設定一個極其宏大且需耗時數月（如半年或一年）才能完成的目標。來源指出，如果等到專案最後才進行測試和交付，萬一初始方向錯誤，這一整年的薪水和開發資源就等於白白浪費。

### 2. 任務切割與持續交付

敏捷思維強調將龐雜的工作拆解成**一到兩週即可完成的小階段**。

- **先做出「可以動」的東西**：即使第一週只做出一個「殼子」或極簡版本，也比完全沒有進度好。
- **每一步皆可 Demo**：開發過程應確保每一個階段性的成果都能進行 Demo、測試與修正，而不是等到最後一步才整合。
- **由小變大**：建議先從小題目做起，再慢慢增加功能，避免一開始就處理過於複雜的架構而導致最終時間不足、品質不佳。

### 3. 高頻率的整合與 AI 協作

在過去，團隊合作可能各自分頭做三週再回來整合，但這往往會導致整合失敗。

- **快速整合**：來源提到，利用現代 AI 編碼工具（如 Cursor），原本繁瑣的整合工作可能在 10 分鐘內就能完成。
- **養成整合習慣**：開發者應利用 AI 的速度優勢，養成更頻繁、更小步快跑的整合習慣。

### 4. 開發者角色的轉變：想法大於編碼

隨著 AI 協作工具的出現，程式開發已達到近百倍的加速。

- **批判性思考 (Critical Thinking)**：開發者不應只是被動地接受題目並「照單全收」，而應學習進行批判性思考與評估。
- **核心價值在於想法**：AI 處理了大部分工程上的「苦工」，因此開發者的價值在於「有想法」。如果缺乏主見與想法，工程師在 AI 時代將面臨極高的被取代風險。

### 5. 勇於溝通與反饋

敏捷開發也包含一種對反饋的開放態度。來源鼓勵開發者在看到題目或計畫時就提出意見，而不是等到最後拿到成績或結果不理想時才抱怨。這種即時的雙向溝通能確保專案方向不偏離預期。

---

**比喻理解：**敏捷開發就像是「捏陶土」而非「蓋大樓」。傳統蓋大樓必須先打好巨大的地基，幾個月後蓋好結構才能看見樣貌，一旦發現蓋歪了就很難修改。而敏捷開發是先捏出一個粗糙的小泥偶（MVP），每幾分鐘就捏塑一次細節並觀察好不好看，隨時可以捏掉重練或微調，最終在不斷的微型修正中完成精美的作品。

什麼是 ORM？它在資料庫操作中扮演什麼角色？

請比較 JavaScript 中同步與非同步溝通的差異。

package-lock.json 檔案的主要用途為何？