

這份教學原始碼摘錄自國立臺灣大學「網頁程式設計」課程，主要探討從**前端開發過渡到後端與全端應用的關鍵轉型期**。課程核心在於建立正確的開發心態與協作技術，特別強調**作業並非區分程度高下**，而是為了讓學生在面對**難度非線性提升**的後端複雜度前，透過實作來內化技術細節並精進**程式重構與模組化**的能力。

在技術實務方面，講師詳盡解析了**專業版本控制流 (Git Flow)** 的運作邏輯，指導學生如何利用 **Fork**、**Branch** 與 **Pull Request** 參與開源專案。教學中特別點出使用 **Rebase** 取代傳統 **Merge** 的重要性，旨在維持清晰的提交歷史，並解決多人在開發過程中的**程式碼衝突 (Merge Conflict)**。

針對課程評分與學生焦慮，講師提出了深刻的教育理念，呼籲學生應**放下對成績的執著**，轉而追求創造出能服務群眾、令他人肯定的高品質作品。他強調**學習的快樂是無價的**，並鼓勵學生站在使用者角度思考，讓成績成為認真投入後自然產生的**水道渠成之結果**。

這份教學原始碼主要探討**全端網頁開發**的轉型與後端基礎邏輯，強調在AI輔助編程的時代，工程師必須掌握底層細節才能驅動複雜的**客製化需求**。課程首先說明如何將前端專案拆解為**前後端分離**的架構，並透過 **HTTP 協定**與 **RESTful API** 的設計規範，確保資料傳輸的精確性與一致性。文中深入解析了 **Node.js 執行環境**與 **Express 框架**的核心機制，特別是利用**中介軟體 (Middleware)** 定義請求處理流程與生產線的概念。最後，作者介紹了 **環境變數 (.env)** 在安全管理敏感金鑰上的重要性，並示範如何引導 AI 根據前端 UI 邏輯來規劃合理的**後端 API 與資料庫模式**。

Discuss 作業與評分制度

這門課程的作業與評分制度設計核心在於**實踐與學習過程**，而非單純的技術高低區分。以下根據來源內容，為您詳細分析作業要求、評分機制及老師的評分哲學：

一、作業制度與要求

這門課總共有 **7 個作業**，約佔總成績的 **50% 至 60%**。

- 作業目的：** 老師強調作業的主要目的是讓學生**練習課堂所學**，將聽過的技術轉化為實際操作。作業設計並非為了區別程度高下，而是希望學生能透過實作發現理解上的落差。
- 難度變化：** 作業一與作業二相對簡單，學生甚至可以依賴 AI 生成程式碼而不需完全理解技術細節。然而，從作業三開始，**難度將呈現「非線性」提升**，並增加更多結構化的要求，如將程式碼拆解為 component、hook 或 context 等模組化設計。
- 高標自我要求：** 老師建議學生不要「做完就好」，應主動改進作品。雖然額外的優化對成績幫助有限（因為大家分數普遍都很高），但這個過程有助於真正理解技術背後的意義。
- 最終目標：** 後續作業（如作業 4 到 7）會涵蓋第三方 API 整合（如 Google API）、社群應用（多對多溝通）、AI 聊天機器人整合及專案部署，並要求學生從使用者回饋中進行改進。

二、評分制度與同儕審查 (Review)

本課程採用**同儕審查 (Peer Review)** 與助教/老師評分並行的制度。

• **審查的主觀性與客觀性：** 許多學生反應審查成績主觀，但老師認為主觀意見若能代表眾人的看法，即具有**客觀意義**。

• **評分統計：** 根據目前統計，作業的前兩個最高分平均大於 **5.2 分**（滿分 6 分）。在 150 多位同學中，評分差距達到 4 分的個案僅有 9 位，顯示大多數人的評分標準相當接近且寬容。

- **申訴機制：**若學生自認被冤枉（例如審查者完全看不懂你的作品），仍可提出申訴。
- **期末專題評分：**期末專題會由老師與三名助教進行四重評分。老師們會花費兩週時間審查 100 多組作品，並針對評分差距較大的個案開會討論，以達成公平共識。

三、成績分佈與心態建議

老師對成績有一套獨特的觀察與哲學：

- **海嘯式分佈 (Tsunami Distribution)：**課程成績呈現「海嘯式」分佈，中位數通常很高（約 85 至 87 分以上），但平均值會被後段放棄或未交作業的同學拉低到 80 至 82 分左右。
- **放下成績執著：**老師誠摯呼籲學生專注於「**學習的快樂**」與「**產出作品的成就感**」，而非計較一兩分的得失。只要認真投入，成績自然會是水到渠成的結果。
- **停修建議：**如果學生對於成績極度執著（非拿到 A+ 不可），老師建議考慮停修。因為這門課的審查制度包含不可控的人為因素，且難度持續攀升，如果不認同這種不確定性，學習過程會非常痛苦。

四、專案協作與期末構思

- **期末點子 (Final Project Idea)：**學生需提交期末計畫構想，只要提交即可獲得 **2 分**。老師會匿名分享這些點子，鼓勵大家集思廣益。
- **Git 協作規範：**為了確保多人協作時程式碼不會損壞，學生應練習使用 Git 的 **Branch (分支)** 與 **PR (Pull Request)** 流程，這也是專業開發者的必備素養。

總結來說，這門課的作業與評分制度是為了培養具備**實戰能力與群眾思維**的開發者。老師希望學生能從「為了成績而努力」轉變為「為了打造出好服務而努力」。

您可以將這套評分制度想像成一場**馬拉松比賽**：老師並不要求每個人都要拿第一名，但希望每位選手都能享受跑動的過程，並在終點線前展現出自己精心打磨的體態與成果；而評審就像是路旁的觀眾（同儕），他們的掌聲（分數）雖然帶有個人情感，但整體的歡呼聲通常能真實反映出運動員的努力程度。

為什麼作業建議使用 Git 開支（branch）進行開發？

根據 RESTful API 規範，如何正確定義資源與動詞？

在 GitHub 上參與開源專案的 Fork 與 PR 流程為何？

Discuss 同儕審核主觀性.

關於**同儕審核 (Peer Review) 的主觀性**，根據提供的來源，這門課程的教師與學生之間有過多次討論。以下是從來源中整理出的關鍵觀點：

1. 主觀性是評核的本質

來源指出，無論是同儕審核、助教審核，甚至是 AI 審核，**本質上都是主觀的**。教師認為「主觀」這兩個字並非原罪，關鍵在於**當個人的主觀意見能夠代表大眾的客觀意見時，主觀就不再是問題**。如果多數人看過後給出的評價都相近，那麼這種主觀認知的結果其實具有參考價值。

2. 統計數據顯示的穩定性

雖然學生擔心主觀性會導致成績不公，但來源提供的統計數據顯示，同儕評分其實相對穩定且慷慨：

- **評分趨於一致**：在 150 幾位同學中，僅有 9 位的兩個評論分數差距達到或超過 4 分；整體的平均分數差距僅約 0.5 分，顯示大家的意見相當接近。
- **平均分數偏高**：前兩個最高分的平均值通常大於 5.2 分（滿分可能為 6 分），這顯示同儕在評分時多半持包容與鼓勵的態度。
- **排除極端值**：目前的制度採取「三取二」的方式，即剔除一個最低分後計算平均，這能有效減少遇到「極端主觀」評分者的影響。

3. 同儕審核作為學習的一環

教師強調，程式設計（尤其是 Web Programming）的服務對象是**群眾**。因此，同儕審核具有以下教育意義：

- **練習站在使用者角度思考**：如果創作者自覺作品很棒，但多位同儕卻無法理解或不知道如何使用，這本身就是一個需要檢討的問題。
- **校準認知標準**：當學生的成績在剔除低分後仍不理想，這是一個契機去思考自己對「好作品」的認知是否與大眾不同。
- **接受不可控因素**：在實際開發中，別人的回饋是不可能完全掌握的，學習如何應對這種不確定性也是課程的一部分。

4. 公平性與申訴機制

儘管存在主觀性，課程仍設有配套措施以確保公平：

- **申訴管道**：如果學生真切認為自己被冤枉（例如同儕完全沒看懂作品），仍然可以提出**申訴**。
- **專業評分介入**：對於期末專題（Final Project），則會由三名助教與教師共四人進行評分，並會針對分數落差較大的個案開會討論，以達成合理的共識。

教師最終建議學生應**放下對成績的執著**，將重心放在學習的過程與創作的快樂上，因為同儕審核的目的是為了練習與理解，而非單純的技術區分。

類比說明：同儕審核就像是**街頭試吃活動**，雖然每個人的口味（主觀）不同，但如果大多數人都覺得太鹹，那這道菜的調味（作品）客觀上確實有調整的空間，而這正是創作者需要獲得的真實回饋。

為什麼 Gitflow 建議從 developer 分支開新功能？

根據統計，同儕審核分數差距達四分的學生有多少位？

什麼是 HTTP 狀態碼中的 400 與 500 錯誤？

Discuss Git 協作流程。

在軟體開發中，**Git 協作流程**是確保團隊成員能平行開發且不互相干擾的核心機制。根據來源內容，這套流程不僅包含技術操作，更涉及專案管理的邏輯：

一、分支管理策略 (Git Flow)

來源強調，開發者應維持一個隨時可以拿出來 **demo 且穩定的 main (或 master**)** 分支**。

1. **開發分支 (Developer Branch)**：每位工程師應從 `developer` 分支（而非 `main`）切出功能分支進行開發。`developer` 分支本身應是相對穩定的，包含最新開發的功能，並經過個人測試。
2. **功能開發與併回**：在自己的分支完成開發且確認無誤後，再併回 (`merge`) 到 `developer` 分支。
3. **發布流程 (Release Flow)**：當 `developer` 分支穩定後，會切到 `release` 分支進行最終 QA 檢查，確認完全沒問題後才併入 `main` 分支進行發布。這能有效避免直接在 `main` 上更動導致系統「爛掉」而無法交作業或展示。

二、參與開源專案與 Forking 流程

當你要參與他人或開源專案時，通常無法直接對原專案 (Upstream) 進行 `push`。

1. **Fork 專案**：在 GitHub 網頁上按「Fork」，這會在你的帳號下建立一個**鏡像倉庫 (Mirror Repo)**，這個你擁有的遠端倉庫通常被稱為 `origin`。
2. **Clone 與設定 Upstream**：將 Fork 回來的專案 `clone` 到本地端後，需手動設定**上游遠端倉庫 (**upstream**)**。指令為 `git remote add upstream <原作者URL>`，這樣才能隨時跟隨原作者的更新。
3. **同步更新**：老師強烈建議使用 **rebase** 取代 `merge` 來進行同步。使用 `git pull --rebase upstream main` 可以確保你的本地開發基礎 (`base`) 是建立在原專案最新的狀態上，這能避免產生混亂的合併紀錄 (`history`)。

三、發送 Pull Request (PR) 的步驟

這是在團隊或開源協作中，請求原作者接受你程式碼的標準做法：

- **本地開發**：在本地建立新的功能分支進行修改，避免直接改動 `main`。
- **推送到 Origin**：修改完成後，先 `push` 到你自己的遠端倉庫 (`origin`)。
- **發起 PR**：到 GitHub 頁面點擊「New Pull Request」，請求將你的分支合併到 `upstream` 的 `main`。
- **代碼審查 (Code Review)**：原作者會收到通知並檢查你的程式碼，確認沒問題後才會同意合併。

四、協作中的溝通與安全規範

1. **使用 GitHub Issues**：老師建議團隊溝通技術細節（如 API 規格變動）時，**不要只在 LINE 或 Slack 講**，而應該開一個 **Issue** 記錄。這能讓所有成員追蹤工作進度（Open/Close 狀態），並留下歷史紀錄供未來查詢。
2. **保護機敏資訊**：在協作中，絕對不要將 API Key 等敏感資料 `push` 到 Git 上。應將這些資料放在 `.env` 檔案中（並加入 `.gitignore`），同時提供一個 `.env.example` 範本，讓夥伴知道需要設定哪些環境變數。

總結來說，Git 協作流程就像是一場**接力賽**：每位隊員都在自己的跑道（Branch）上努力，跑完後需要經過裁判（Reviewer）確認交棒（PR），最後所有人的努力才會匯集成完整的成績；而 `rebase` 就像是確保每個人在出發前，都已經站在目前最新的起跑線上，讓整場比賽的紀錄清晰且順暢。

簡述 Git 中 `fetch` 與 `pull` 的主要差異。

為什麼建議使用 `rebase` 取代 `merge` 來整合分支？

在 Express 框架中，Middleware 的主要作用與運作流程為何？

Discuss 分支管理與 Rebase.

在軟體開發協作中，**分支管理 (Branch Management)** 與 **Rebase** 是確保代碼穩定性及維護整潔歷史紀錄的兩大核心技術。根據來源內容，以下為詳細的討論：

一、分支管理策略 (Git Flow)

分支管理的根本目的在於實現**平行開發**，同時確保主線代碼的品質。

1. **維護穩定的 Main 分支：** `main` (或舊稱 `master`) 分支應始終保持在隨時可以演示 (demo) 且無 bug 的狀態。學生不應直接在 `main` 上進行開發，以避免因遇到無法解決的 bug 而導致整個專案「爛掉」無法交作業。
2. **開發流程：**
 - **開發者分支 (Developer Branch)：** 這是一個眾人開發的基礎分支，應保持相對穩定並經過單元測試。
 - **功能分支 (Feature Branch)：** 每位工程師應從 `developer` 分支切出獨立分支來開發特定功能。
 - **發布分支 (Release Branch)：** 在合併回 `main` 之前，通常會先進入 `release` 分支進行最後的 QA 檢查。
3. **小規模協作：** 對於只有兩三個人的小型團隊，`developer` 與 `main` 可以合併，但仍應保有「開分支開發、確認無誤再合併」的觀念，而非直接覆蓋代碼。

二、Rebase 的意義與優勢

在合併分支時，老師強烈建議使用 **rebase** 來取代傳統的 **merge**。

1. **保持歷史紀錄整潔：** `merge` 指令會將多個分支的更動交錯在一起，導致歷史紀錄 (commit history) 變得混亂。而 `rebase` 則像是在最新的基礎上「插隊」，讓歷史紀錄呈現一條直線，易於追蹤。
2. **本地解決衝突：** 使用 `merge` 往往是把衝突交給對方或主分支處理；而 `rebase` 則是要求開發者在將代碼推送到遠端前，先自行在本地解決與最新版本的衝突。
3. **同步最新基礎：** 當遠端分支 (如 `upstream`) 有更新時，透過 `git pull --rebase` (等同於 `fetch` 加上 `rebase`)，可以確保你的開發基礎是建立在最新的代碼之上。這能避免在發送 Pull Request (PR) 時，因為基礎版本過舊而遭到系統拒絕。

三、協作與同步流程 (以 Fork 為例)

參與他人專案或開源專案時，分支與同步的邏輯更為嚴謹：

- **建立上游連結：** 需將原作者的倉庫設定為 `upstream`。
- **同步步驟：** 應養成習慣，隨時將 `upstream` 的 `main` 分支 `fetch` 回來，並對自己的開發分支進行 `rebase`。
- **PR 發送：** 先將本地分支 `push` 到自己的遠端倉庫 (`origin`)，再從 GitHub 介面發起 PR 到原專案的 `main`。
- **自我更新：** 若要讓自己的 `origin main` 與原作者同步，需先在本地 `local` 完成更新 (透過 `fetch` 與 `rebase`)，再將更新後的本地 `main` 分支 `push` 回自己的 GitHub。

總結來說，分支管理提供了一個安全的操作空間，而 Rebase 則是一種負責且優雅的對齊方式。

您可以將這套機制想像成興建一棟大樓：main 分支就是已經完工且驗收通過的樓層（基礎），而每個 feature 分支就像是在旁邊搭建的臨時腳手架進行加蓋作業。Rebase 的過程就像是當大樓地基又加高時，您必須先將自己的腳手架也墊高到同樣的水平，並確認結構依然穩固後，再將新的工程併入大樓本體，這樣整棟建築的結構紀錄才會是清晰且連續的。

比較 Git Fetch 與 Git Pull 的差異。

何謂 RESTful API 的設計風格？

說明 Node.js 中 Middleware 的基本概念。

Discuss 後端開發基礎.

後端開發是建構現代網路應用程式的核心，負責處理資料邏輯、資料庫管理及伺服器運算。根據來源內容，後端開發的基礎可以從技術棧、通訊協議、開發環境與開發流程四個維度來探討：

一、後端技術棧與核心工具

後端開發涉及多層次的技術整合，從執行環境到資料驗證都有其專屬工具：

1. **Node.js**：這是一個基於 Chrome V8 引擎的 **JavaScript 執行環境 (Runtime Environment)**，其目標是讓 JavaScript 可以在瀏覽器以外的任何地方執行。它的特性是處理**非同步 (Asynchronous) I/O**，能有效應對高併發的網路請求。
2. **Express**：這是一個輕量級的 **Web 應用程式框架**，建立在 Node.js 之上。其核心概念是 **Middleware (中介層)**，開發者可以將後端邏輯定義為一套「生產線」或「工作流程」，將前端的 Request 逐步處理後轉化為 Response。
3. **資料庫與 ORM**：後端通常需要連接資料庫（如 SQL 或 NoSQL）來儲存資料。為了更方便地溝通，開發者會使用如 **Prisma** 這樣的 ORM 工具，以及如 **Zod** 進行資料驗證。
4. **安全與驗證**：包含帳號管理、密碼加密（Bcrypt）及身分驗證（JWT）等技術，確保資料交換的安全性。

二、通訊協議：HTTP 與 RESTful API

後端與前端溝通的語言即是 **HTTP 協議**，這是一種「請求 (Request) 與回應 (Response)」的機制。

- **HTTP Request 結構**：包含 **Request Line**（動詞與網址）、**Header**（元數據，如格式或授權碼）及 **Body**（實際傳輸的訊息內容）。
- **標準動詞 (Methods)**：
 - **GET**：取得資源（如拿菜單）。
 - **POST**：提交資料（如點餐）。
 - **PUT/PATCH**：更新資源。
 - **DELETE**：刪除資源。

- **狀態碼 (Status Codes)**： **200** 代表成功，**404** 代表前端網址打錯 (Client 錯誤)，**500** 代表伺服器出問題 (Server 錯誤)。

- **RESTful Style**：這是一種建議的 API 設計風格，強調**資源由 URI 決定**，並使用標準的 HTTP 動詞來操作資源，避免將動詞與名詞混淆。

三、開發環境與安全性

1. **Localhost 開發**：在開發階段，前端與後端通常都跑在同一台電腦的 **localhost**，但會分配不同的 **Port (門牌號碼)** (例如前端 3000，後端 3001) 以避免衝突。最終部署時才會搬移到雲端環境。
2. **環境變數 (.env)**：為了保護機敏資料 (如 API Key)，後端開發者會將密鑰存在本地的 **.env** 檔案中，並在 Git 中忽略此檔，僅上傳 **.env.example** 作為範本。

四、開發流程與 AI 協作

1. **API 導向設計**：前後端溝通的關鍵在於 **API 規格**。開發者應先根據 UI/UX 需求定義 API，並建議使用 **GitHub Issues** 記錄溝通與修改細節，而非僅透過社群軟體討論。
2. **AI 的角色**：在 AI 時代，後端工程師不應只是複製程式碼，而要具備**駕馭 AI 的能力**。AI 可能會生成冗長且雜亂的單一檔案，工程師必須主動要求進行 **Refactor (重構)**，將程式碼模組化 (Component, Hook, Context)，以維持架構的純淨與可維護性。
3. **逐步實作**：建議從影響較大的部分開始設計 (前端優先或後端優先)，並確保伺服器隨時處於可運行的狀態，避免一次性開發過多複雜功能導致無法驗證。

總結來說，後端開發基礎不只是寫程式碼，更是關於如何建立一套穩定、精確且安全的「規則」，讓資料能在正確的時間點流動到正確的地方。

您可以將後端開發想像成**餐廳的廚房作業系統**：**HTTP** 是服務生傳達的需求 (菜單與指令)；**Node.js 與 Express** 是廚房的運作規則與生產線；而**資料庫**就是冰箱與儲藏室。後端工程師的職責就是確保廚房能精確解讀客人的點單 (API)，並從冰箱拿出正確的食材 (資料)，經過標準流程處理後，端出一道符合預期的料理 (Response)。

為什麼作業三建議將 CSV 檔案從前端移至後端？

在 Express 框架中，Middleware 的核心功能是什麼？

根據 RESTful 風格，HTTP 動詞與資源路徑應如何設計？

Discuss 後端開發架構。

根據來源內容，**後端開發架構**不僅涉及技術選型，還包含通訊協議、邏輯層級的組織以及開發流程的規劃。以下詳細討論後端架構的核心組成與設計原則：

一、後端技術棧與運行環境

後端架構的基礎建立在提供穩定邏輯與資料服務的環境上。

1. **Node.js 運行環境**：Node.js 並非程式語言，而是一個基於 Chrome V8 引擎的 **JavaScript 運行環境 (Runtime Environment)**。它允許 JavaScript 在瀏覽器以外的地方執行，具備處理跨平台、**非同步 I/O** 的能力。

2. **Express 框架**：為了避免直接撰寫底層低階程式碼，開發者通常使用 **Express** 框架。它簡化了後端邏輯的實現，核心概念是提供一個高階的開發環境來處理伺服器邏輯。
3. **資料庫與 ORM**：後端架構通常包含資料庫（如 SQL 或 NoSQL）來管理使用者資料。為了更方便地進行溝通，會導入 **Prisma** 等 ORM (Object-Relational Mapping) 工具，以及 **Zod** 進行資料驗證。

二、Middleware (中介層) 的生產線架構

Express 的核心架構在於 **Middleware**，這是一種將伺服器功能模組化的方式。

- **工序化的服務流程**：Middleware 就像一條**生產線**或**工作流程**，它接收來自前端的 Request，逐層處理後產生 Response 並回傳。
- **鏈式處理**：每個中介層 function 通常包含 `req` (request)、`res` (response) 與 `next` 三個參數，透過呼叫 `next()` 將處理權交給下一個環節，直到完成整個 API 服務。
- **路由 (Routing) 管理**：伺服器會根據不同的路徑 (Route) 將請求分派到對應的 Middleware 處理。架構設計上建議將 API 邏輯從主檔案抽離，以維持程式碼的簡潔。

三、API 設計規範：RESTful Style

後端架構的對外接口主要遵循 **RESTful** 風格，這是一種建議的 API 設計風格而非強制規範。

- **以資源為中心**：資源應由 **URI** 決定，例如 `/api/user/123` 代表特定使用者資源。
- **標準 HTTP 動詞**：精確使用 **GET** (取得)、**POST** (新增)、**PUT** (更新資源)、**PATCH** (局部修改) 與 **DELETE** (刪除) 等動詞來操作資源，避免將動詞混入 URI 中。
- **狀態碼回饋**：伺服器應回傳精確的 **HTTP Status Code**，如 200 (成功)、404 (客戶端網址錯誤) 或 500 (伺服器錯誤)。

四、開發環境與安全性架構

1. **環境變數 (.env) 隔離**：為了保護機敏資訊（如 API Key），架構上會使用 **.env** 檔案儲存私密金鑰，並透過 `.env.example` 提供範本供他人協作。
2. **前後端分離開發**：在開發階段，前端與後端通常各自運行在 **localhost** 的不同 **Port** (門牌號碼)，例如前端 3000、後端 3001。
3. **模組化與重構 (Refactoring)**：好的架構不應將所有程式碼塞入單一檔案。開發者應主動將程式碼拆解為 **component**、**hook** 或 **context** 等模組，避免產出數千行難以維護的代碼。

五、實作計畫與 AI 協作

當從純前端轉向全端架構時，建議從影響較大的部分（如 UI/UX 需求或第三方服務限制）開始設計 API 與 Database Schema。利用 AI 可以輔助生成資料表結構與 API 定義，但開發者必須具備**駕馭 AI 的能力**，審視其產出的合理程度並進行調教。

總結來說，後端開發架構就像是一座**全自動化工廠**：**Node.js** 是工廠的電力與地基，**Express** 是生產設備的控制台，而 **Middleware** 則是工廠內一條條精密設計的傳送帶（生產線），確保每一份原料（Request）都能經過標準化的工序轉化為成品（Response）並送到客戶手中。

為什麼建議開發者使用 `rebase` 而非 `merge` 同步程式碼？

請解釋 HTTP 狀態碼中 400 與 500 系列的差異。

在開發開源專案時，`.env` 檔案的主要功能為何？

Discuss HTTP 協定規範.

HTTP (Hypertext Transfer Protocol) 是網路通訊的核心協定，定義了客戶端 (Client) 與伺服器 (Server) 之間溝通的語言與文法。根據來源內容，HTTP 協定的規範可從請求結構、操作動詞、狀態碼及設計風格四個面向來討論：

一、HTTP 請求與回應的結構

HTTP 的運作機制是基於「**請求 (Request)** 與**回應 (Response)**」的循環。

• 請求結構 (Request Structure)：

1. **Request Line**：包含操作動詞 (Method)、資源路徑 (URI) 及協定版本。
2. **Header**：包含元數據 (Metadata)，如資料格式 (Content-Type) 或授權資訊 (Authorization)。
3. **Message Body**：實際傳輸的資料內容，例如 JSON 或 HTML。

• 回應結構 (Response Structure)：

1. **Status Line**：包含協定版本、**狀態碼 (Status Code)** 以及原因描述。
2. **Header 與 Body**：與請求結構類似，回傳伺服器處理後的資料與相關資訊。

二、標準操作動詞 (HTTP Methods)

為了達成精確的溝通，HTTP 定義了多個動詞來代表不同的操作意圖：

- **GET**：取得資源（如開啟網頁、拿取菜單）。
- **POST**：提交資料以進行處理（如點餐、送出表單）。
- **PUT**：更新或創建資源（強調完整的替換）。
- **DELETE**：刪除指定的資源。
- **PATCH**：對資源進行局部修改（如在已送出的訂單中加點東西）。
- **HEAD**：與 GET 類似，但僅回傳 Header，不含 Body。

三、狀態碼 (Status Codes)

伺服器透過狀態碼告知客戶端處理結果，這些代碼分為五大類：

- **2xx (成功)**：最常見的是 **200 OK**，代表請求已成功處理。
- **3xx (重定向)**：代表請求需要進一步操作（如跳轉網址）才能完成。
- **4xx (客戶端錯誤)**：代表客戶端發送的請求有誤，例如 **404 Not Found**（網址打錯）。
- **5xx (伺服器錯誤)**：代表伺服器端發生故障，例如 **500 Internal Server Error**。

四、RESTful API 設計風格

來源強調，雖然網路協定具備一定的容錯性（Forgiving），但優良的開發者應遵循 **RESTful** 風格以維持溝通的精確性。

1. **資源由 URI 決定**：URI 應代表資源（名詞），而不應包含動詞。
2. **動詞與名詞分離**：例如應使用 GET /questions 來獲取問題清單，而非自創 /getQuestions 這樣的網址。
3. **結構化路由**：利用斜線與 ID 進行層級化管理（如 /questions/123 代表特定 ID 的問題）。

總結來說，遵循 HTTP 協定規範是後端開發的基礎，這能確保前後端系統能以標準化的方式交換資料，減少開發中的誤解。

您可以將 HTTP 協定想像成**餐廳的點餐流程**：**URL** 是菜單上的編號；**Method** 是您的動作（是要看菜單、點菜還是退冰水）；而**狀態碼**則是服務生的回覆（「沒問題馬上去辦」是 200，「抱歉沒這道菜」是 404，「廚房失火了」則是 500）。

為什麼建議開發時優先從前端或後端擇一開始？

在 Git 協作中，使用 rebase 取代 merge 的主因為何？

根據 RESTful 風格，URI 的定義應遵循什麼原則？

Discuss Node.js 運行環境。

Node.js 是現代全端開發的基石，它並非一種程式語言或簡單的套件，而是一個基於 Chrome V8 引擎的 **JavaScript 運行環境 (Runtime Environment)**。以下根據來源內容，詳細討論 Node.js 的核心特性、架構及其在開發中的角色：

1. 核心定義與起源

Node.js 由 Ryan Dahl 於 2009 年開發，其核心目標是讓 **JavaScript 脫離瀏覽器的限制**，在伺服器端、終端機或其他任何地方執行。

- **V8 引擎**：Node.js 採用了 Chrome 瀏覽器開源的 **V8 引擎**，負責將 JavaScript 程式碼轉譯並執行。
- **系統級能力**：作為一個完整的運行環境，它提供了類似作業系統的功能，包含**檔案系統 (File System)**、**網路通訊 (Networking)**、**二進位資料處理及加密**等能力。

2. 非同步 I/O 與 libuv

Node.js 最顯著的技术優勢在於處理**非同步 (Asynchronous) I/O**。

- **解決效能落差**：CPU 的執行速度極快（以 GHz 計），但與資料庫或網路通訊拿取資源則相對緩慢（需耗費毫秒級時間）。
- **libuv 庫**：Node.js 透過內部的 libuv 函式庫來處理跨平台的非同步 I/O，確保程式在等待資料回傳時不會發生阻塞 (Blocking)，能持續處理其他請求，這對於高併發的後端服務至關重要。

3. NPM 生態系統

Node.js 的強大在於其極其活躍的社群與 **NPM (Node Package Manager)**。

- **模組化開發**：開發者可以利用 NPM 管理成千上萬由社群貢獻的開源模組，從簡單的工具到複雜的框架（如 Express）。
- **指數型成長**：隨著社群化與多對多應用需求的增加，NPM 的套件數量呈現爆量增長（已超過 200 萬個）。
- **資源消耗**：來源幽默地將 NPM 比喻為「黑洞」，因為每個專案往往需要下載龐大的 `node_modules` 到本地端，消耗大量硬碟與網路資源。

4. 版本管理與穩定性

Node.js 維持著非常規律的更新節奏，大約每半年（4 月與 10 月）發布一次新版本。

- **LTS (Long Term Support)**：為了開發穩定性，建議使用**偶數版號**的 LTS 版本（如 v22），這代表該版本具備長期維護支援。
- **開發建議**：若本地環境仍停留在過舊的版本（如 v14 或 v18），應盡快更新至最新的 LTS 版本。

5. 開發實務與環境設定

在實際操作中，Node.js 可以直接透過指令碼執行 JavaScript 檔案（如 `node test.js`），不需開啟瀏覽器。

- **ES Module (ESM)**：現代開發常使用 ESM 格式，若在執行時遇到警告訊息，可透過將副檔名改為 **.mjs** 或在 `package.json` 中設定 `"type": "module"` 來解決。
- **後端與前端隔離**：在開發全端應用時，前端（如 React）與後端（如 Express）雖然都跑在 Node.js 環境下，但應分配不同的 **Port**（如 3000 與 3001）以避免衝突。

總結來說，Node.js 為 JavaScript 賦予了處理伺服器邏輯與硬體資源的能力，使其從單純的網頁腳本變成了強大的通用開發工具。

您可以將 Node.js 運行環境想像成一個**萬能的翻譯官與後勤基地**：它不僅精通 JavaScript 這門語言（V8 引擎），還隨身帶著整套工具箱（檔案與網路系統），能讓本來只能在溫室（瀏覽器）裡生長的植物（程式碼），成功移植到各種嚴苛的戶外環境（伺服器）中茁壯成長。

簡述 RESTful API 的設計風格與其建議規範。

說明 Gitflow 中各開發分支（Branch）的用途。

解釋 Node.js 與 Express 兩者之間的關係。

Discuss Express 框架應用。

Express 是一個建立在 Node.js 之上的 **Web 應用程式框架 (Framework)**，其核心目標是包裝底層複雜且低階的溝通細節，讓開發者能以更高階、更易讀的方式實現後端伺服器的邏輯。

以下根據來源內容，詳細探討 Express 框架的應用與核心機制：

1. 核心概念：中介層 (Middleware)

Express 的設計哲學圍繞著 **Middleware (中介層)** 展開，這是理解該框架的關鍵：

- **生產線模型**：老師將 Middleware 比喻為一條「生產線」或「工作流程」。它負責接收來自前端的 Request，經過一系列處理後，最終產生 Response 回傳給使用者。
- **三大參數**：每個 Middleware 函式通常會接收三個參數：`req` (Request)、`res` (Response) 與 `next`。
 - `req`：包含前端傳來的資訊，如 Request Line、Header 或 Body。
 - `res`：用於處理回傳給前端的資料。
 - `next`：這是一個特殊的函式，用於將處理權交給生產線上的下一個步驟。

2. Express 伺服器的基本流程

一個典型以 Express 為基礎的後端程式（如 `server.js`）通常遵循以下開發流：

1. **初始化**：導入 Express 套件並建立一個 Express 物件（如 `const app = express()`）。
 2. **設定全域服務**：在生產線的最前端註冊通用服務，例如處理跨來源資源共用 (CORS) 或解析 JSON 格式的資料。
 3. **定義路由 (Routing)**：使用 `app.get()`、`app.post()`、`app.put()`、`app.delete()` 等方法來定義不同 API 路徑的服務邏輯。
 4. **錯誤處理 (Error Handling)**：
 - **404 處理**：在路由清單的最末端，通常會設置一個匹配所有路徑的處理器，當前面的路徑都沒匹配到時，回傳「404 Not Found」。
 - **500 處理**：用於捕捉伺服器內部的非預期錯誤。
 5. **啟動監聽**：呼叫 `app.listen()` 並指定連接埠 (Port)，這才是真正將伺服器跑起來的指令。
 6. 路由與 API 實作
- **順序性**：路由的匹配是由上而下按順序進行的，一旦匹配成功且未呼叫 `next()`，流程就會結束。
 - **RESTful 實踐**：在實作點餐 App 等應用時，Express 可以輕鬆對應不同的資源操作，例如用 `GET` 拿取菜單、用 `POST` 提交訂單。
 - **模組化建議**：老師建議不要將所有 API 邏輯塞在同一個檔案中。當專案變大時，應將路由與服務邏輯拆分到獨立的目錄（如 `api/`）下，以維持程式碼的純淨與可維護性。

4. 與 AI 協作開發

在現代開發環境中，可以利用 AI 生成 Express 的基礎框架，但開發者必須具備引導 AI 的能力。AI 可能會生成冗長且結構混亂的代碼，開發者應主動要求其進行**重構 (Refactor)**，將邏輯模組化為 Component、Hook 或 Context，並正確地整合到 Express 的 Middleware 架構中。

總結來說，Express 提供了一個結構化的環境，讓開發者能專注於定義業務邏輯，而不必處理繁瑣的網路協議細節。

您可以將 Express 框架想像成一間**高度自動化的餐廳廚房**：**Node.js** 是廚房的電力與地基，而 **Express** 則是廚房內的**自動化管理系統**。**Middleware** 就像是廚房裡的各個工作站（洗菜、切菜、烹飪、擺盤），客人的訂單 (Request) 會沿著這條生產線移動，每個站點完成任務後按個鈴（`next`）傳給下一站，最後由出餐口將料理 (Response) 送交到客人手中。

簡述 HTTP 請求中的三種主要方法及其用途。

在 Git 協作中，為何建議使用 rebase 而非暴力 push？

軟體開發中，使用 .env 檔案的主要目的為何？

Discuss API 設計原則。

在開發全端應用程式時，**API（應用程式介面）** 是連接前端需求與後端邏輯的關鍵橋樑。根據來源內容，優秀的 API 設計應遵循以下核心原則：

1. 遵循 RESTful 設計風格

來源強調，雖然網路協定具有高度的容錯性（Forgiving），但為了達到精確溝通，建議遵循 **RESTful** 這一種廣為人知的設計風格。

- **資源由 URI 決定：**URI 應代表「資源」（名詞），例如使用 `/questions` 或 `/user/123`，而不應在網址中包含動詞。
- **動詞與名詞分離：**應利用標準的 **HTTP Methods** 來表達操作意圖，避免將動詞混入網址（例如：應使用 `GET /questions` 而非 `/getQuestions`）。
- **結構化路由：**應按照資源分配進行正確的 **Routing**，利用斜線與 ID 區分特定資源（如 `/questions/123` 代表特定 ID 的問題）。

2. 精確使用 HTTP 動詞與狀態碼

為了讓前後端溝通更精確，開發者必須正確選擇對應的「動作」與「回饋」：

- **標準動詞應用：**
 - **GET：**取得資源（如拿菜單）。
 - **POST：**提交資料以進行處理（如點餐）。
 - **PUT/PATCH：**更新資源，前者通常代表完整替換，後者代表局部修改。
 - **DELETE：**刪除特定資源。
- **狀態碼回饋：**伺服器回應時必須包含 **Status Code** 以告知結果。
 - **2xx：**代表成功（如 200 OK）。
 - **4xx：**代表客戶端錯誤（如 404 網址打錯）。
 - **5xx：**代表伺服器內部錯誤（如 500 Server 當機）。

3. 以需求驅動設計 (Demand-Driven Design)

API 的設計順序應考量專案的複雜度來源：

- **UI/UX 驅動：**若服務的介面互動複雜，應先從前端需求定義 API，確保後端提供的資料格式能支持 UI 呈現。

- **避免過度設計 (Overdesign)**：不要試圖一次想清楚所有規格，最好的驗證方式是快速實作並隨時測試整合。
- **考量第三方限制**：若需整合第三方服務（如 Google API），設計時需考量其頻率限制（Rate Limit）或安全規範。

4. 協作管理與安全性

在多人協作的專案中，API 的變動需要嚴謹的管理：

- **文件化與溝通**：API 的規格應被詳細記錄。若有變動，建議在 GitHub 上開 **Issue** 記錄討論過程，而非僅透過社群軟體討論，以避免死無對證。
- **資料驗證與格式**：API 應指定 `Content-Type`（如 JSON），並在後端進行資料驗證（如使用 Zod）。
- **安全性與機敏資訊**：在設計與實作時，API Key 等敏感資訊絕對不可寫死在程式碼中，應存放在 `.env` 檔案中，並透過 `.env.example` 提供格式範本供他人協作。

總結來說，API 設計的核心在於建立一套「標準化」且「語意清晰」的合約。

您可以將 API 設計想像成**郵政系統的地址與服務規範**：**URI** 是精確的門牌號碼（名詞），**HTTP Methods** 是郵差要執行的動作（投遞、取件、撤回），而 **Status Code** 則是收據回條（成功送達、查無此人、郵局失火）。只有地址格式統一且動作明確，整座城市的資訊交換才能順暢無礙。