真・淺談 RESTful API

by

Triton Ho 

大綱

- 真．淺談 HTTP
- 為什麼要重視 API 設計
- 比較傳統與 API 網頁設計
- RESTful API 設計

真・浅談 HTTP

HTTP 簡介

- HTTP 是通訊協定！（請跟我大喊三次）
  - http://www.w3.org/Protocols/
  - http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- 很多東西是建基於 HTTP 的
  - 一般 RESTful 後端
  - SOAP
  - 很多支持 REST 界面的資料庫
  - HTML

HTTP 理念

- 資源（ resource ）+ 方法（ method ）
  - 就跟中文英文一般，名詞 + 動詞
- 非狀態性通訊（ Stateless protocol ）
  - 等一下教 RESTful 時會說明

淺談 HTTP 資源

- 「資源」一定是名詞！
- 動名詞（ Gerunds ）也是名詞一種，例如要銀行轉帳時，應該建立 /MoneyTransfers 這個物件
- 例子：
  - www.abc.com/v1/users/89072
    - 這是說明在 www.abc.com 之下的 user 物件
    - 這 user 的 id 是 89072
  - www.abc.com/v1/MoneyTransfers (POST)
    - 這是建立轉帳的物件，其內容會是從用戶 A 轉錢到用戶 B
  - www.abc.com/v1/TopSecretReport.ps
    - 這是說明在 www.abc.com 之下， TopSecretReport.ps 的檔案

淺談 HTTP method

- 雖然 HTML 只用上了 HTTP 的 GET 和 POST ，但是 HTTP 還有很多常用的 method
- 你可以不遵守大家的約定俗成，只是後果自負
  - 例子：你用 GET 去建立新 user 物件
    - ~~請在原始碼留下你名字、出生日期和時間~~
- 一些資源可能只支持部份 method
  - 例子： /v1/TopSecretReport.ps 應該只有 GET 的

HTTP method-GET

- 就是讀取資源， READONLY 的
- 只有 GET 才應該使用 Query String 的（很重要）
- 如果以物件 id 結尾的，一般是指讀取單一物
  - 例子： /v1/users/89072
- 沒有以物件 id 結尾的，一般是讀取該物件的 Collection
  - 用戶可以在 QueryString 中，加入額外的搜索條件
  - 例子： /v1/users?AgeMax=20

HTTP method-DELETE

- 就是刪掉特定資源
- 例子： /v1/users/89072
  - 指示伺服器把 ID 為 89072 的 user 物件刪掉

HTTP method-POST

- 建立新資源
  - 例子： /v1/users
  - 一般會把資源的 ID 返回給用戶
  - 延伸思考：在不穩定網路下，用戶建立一份資源時，卻發出了二次的 POST ，怎麼辦？
  - 雖然原始 HTTP POST 定義是 non-idempotent ，但是他沒禁止你自行解決 idempotent 問題

HTTP method-PATCH

- 改動資源的部份內容
- 例子：本來 id 是 80972 的 user 的內容是：Name = Susan, Age = 30 ，現在需要把 Age 改成 31 ，所以
  - URL = /v1/users/89072
  - Method = PATCH
  - Request Body = {“Age” : 31}

HTTP method-PATCH（續）

- 是否 idempotent 是看你怎做改動
- 這是 idempotent 的 PATCH
  - SET Age = 31
- 這不是 idempotent 的 PATCH
  - SET Age = Age + 1

HTTP method-PUT

- 傳統 HTTP 定義：以新上傳的內容，覆蓋掉本來的資源
- 不過，很多人會在 PUT 中支援部份改動 (Partial Update)，而不再另開 PATCH method
- 例子： /v1/user/89072

HTTP method-OPTIONS

- 用來查詢某一資源，其能使用的 methods
- 在 cross domain request 時，用戶端會先發出 OPTIONS 查詢的

常用 HTTP 回答碼： 2xx

- 200(OK)
  - 請求成功了，並且把結果傳回
- 204(No Content)
  - 跟 200 相似，不過沒有結果需要返回
  - 一般來說， DELETE 和 PUT 會常用
- 202(Accepted)
  - 伺服器收到請求，並且確定請求是沒問題的
  - 這單純代表收到了工作，不代表工作做完了
  - 一般來說，工作完成後伺服器會主動推送結果給客戶端

常用 HTTP 回答碼： 3xx

- 304(Not Modified)
  - 如果客戶端已經有了某一物件的副本，但是不知道這物件是否最新版本
  - 客戶端會發出 Conditional-GET ，在 HTTP header 中加入 If-Modified-Since: < 某時間 >
  - 如果客戶端版本是最新的，便回答 304 。否則便 200

常用 HTTP 回答碼： 4xx

- 400(Bad Request)
  - 請求的內容有誤，伺服器拒絕執行
  - 例子： Age 這個資料應該是 Integer 的，客戶端卻傳來 ABC123
- 401(Unauthorized)
  - 最簡單來說：請先登入系統
  - 請檢查 HTTP HEADER 的 Authorization
- 403(Forbidden)
  - 跟 400 不同，伺服器明白請求的內容的
  - 不過，其請求內容與商業邏輯矛盾，伺服器拒絕執行
  - 例子：建立一個貓的物件，其 { 心愛食物 = 洋蔥 }

常用 HTTP 回答碼： 4xx（續 1）

- 404(Not Found)
  - 請求的資源不存在
  - 例子： www.abc.com/v1/users/89073 ，而系統並沒有 89073 這用戶
  - 如果是 GET collection 而沒搜查到滿足條件的物件，是應該返回 200 ，而不是 404 的
    - 例子： www.abc.com/v1/users?AgeMin=30&AgeMax=20
- 409(Conflict)
  - 通常跟 optimistic lock 有關
  - 這是指用戶正在改動的物件，已經被別人先改動過了。

常用 HTTP 回答碼： 4xx（續 2）

- 410(Gone)
  - 一個資源曾經存在，但現在不再存在
  - 大部份懶人會用 404
  - 電商東西賣光了是回傳 403，而不是 410
- 451(Unavailable For Legal Reasons)
  - 台灣不適用，大家怎麼罵都是可以的 X D
  - 當你在某強大國家談 8^2 時回傳的
    - 聽說他們的 OS 是 65 bits 的
  - 你再問下去，小心你變成 404 囉

常用 HTTP 回答碼： 5xx

5XX 系列的錯誤，全都是用戶 / 客戶端沒法自行解決的

- 500(Internal Server Error)
  - 一般來說嘛：伺服器程式有 bug ，或是某部份（例如資料庫）當掉了
- 502(Bad Gateway)
  - 一般來說：伺服器端的分流器層（ load balancing tier ）出現問題
- 503(Service Unavailable)
  - 伺服器沒當掉，只是目前太多人使用而繁忙中，所以先直接拒絕這個請求

為什麼要重視 API 設計

為什麼要重視 API 設計

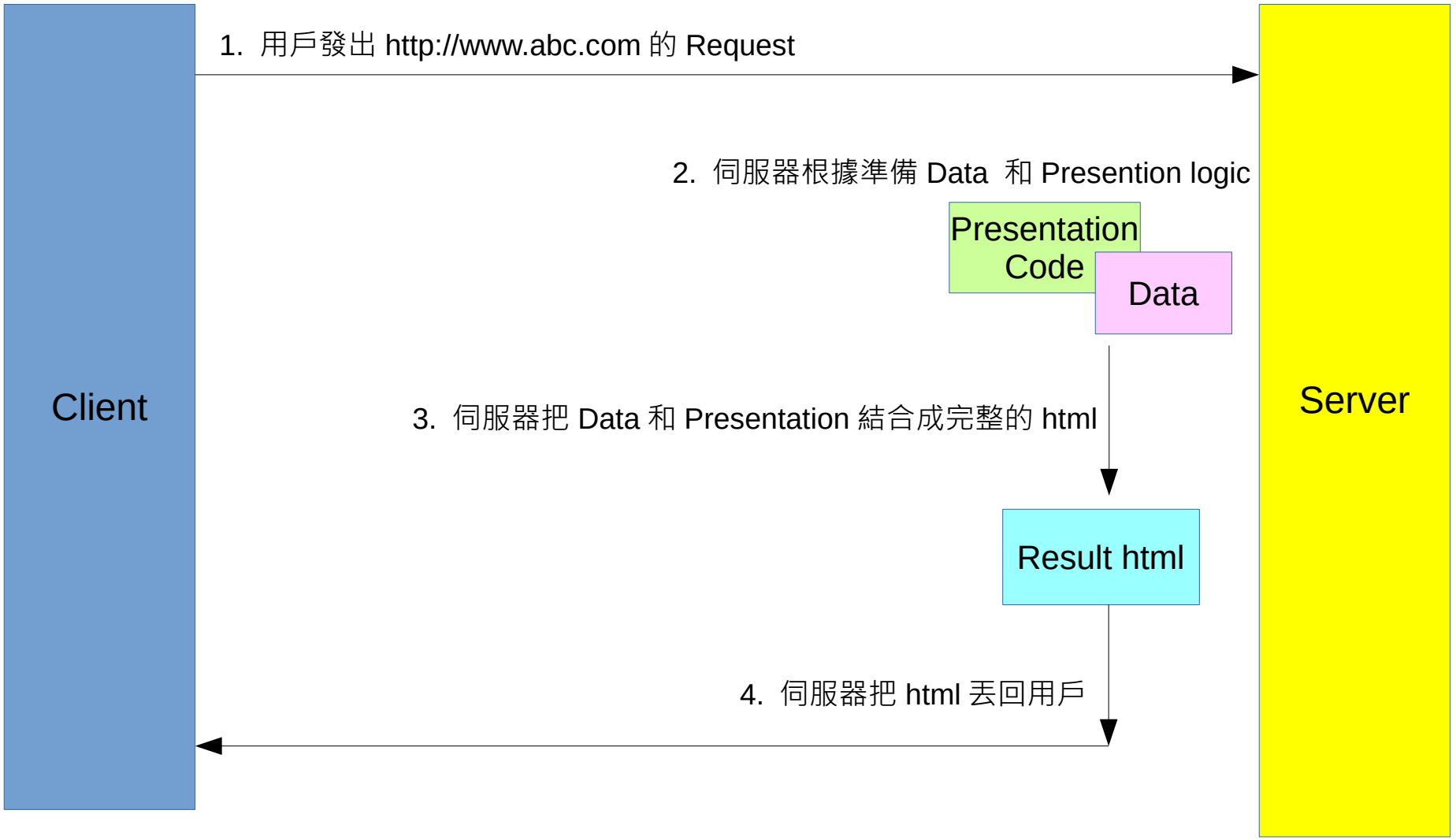
- 改動 API 需要同時改動前端跟後端，現實上會有極大困難
  - 細心的人，應該看到我剛才的 URL 是有 v1 吧
- 錯誤的 API 設計，可以引起：
  - 嚴重的效能問題
  - 扭曲了的程式邏輯，和扭曲了的數據結構
  - 數據錯誤
  - ~~整天在公司加班不陪女友，女友跑掉~~

API 設計重點

- 統一介面（ uniform interface ）是相對上最死不了人的，別為這吵架太久
  - 例子：以下兩個流派都有人喜歡：
    - /v1/pets/1234
    - /v1/users/89072/pets/1234
- 好的 API 永不限制前端設計，即使要輕度違反 uniform interface 原則，一切以使用者感受為最大優先
  - 例子：在頁面中，用戶能同時建立其用戶資料，還有其所有的貓群的。所以，需要建立特殊的 UserAndPet POST
- 再次強調：所有的 API 一定是名詞 / 動名詞

比較傳統與 API 網頁設計

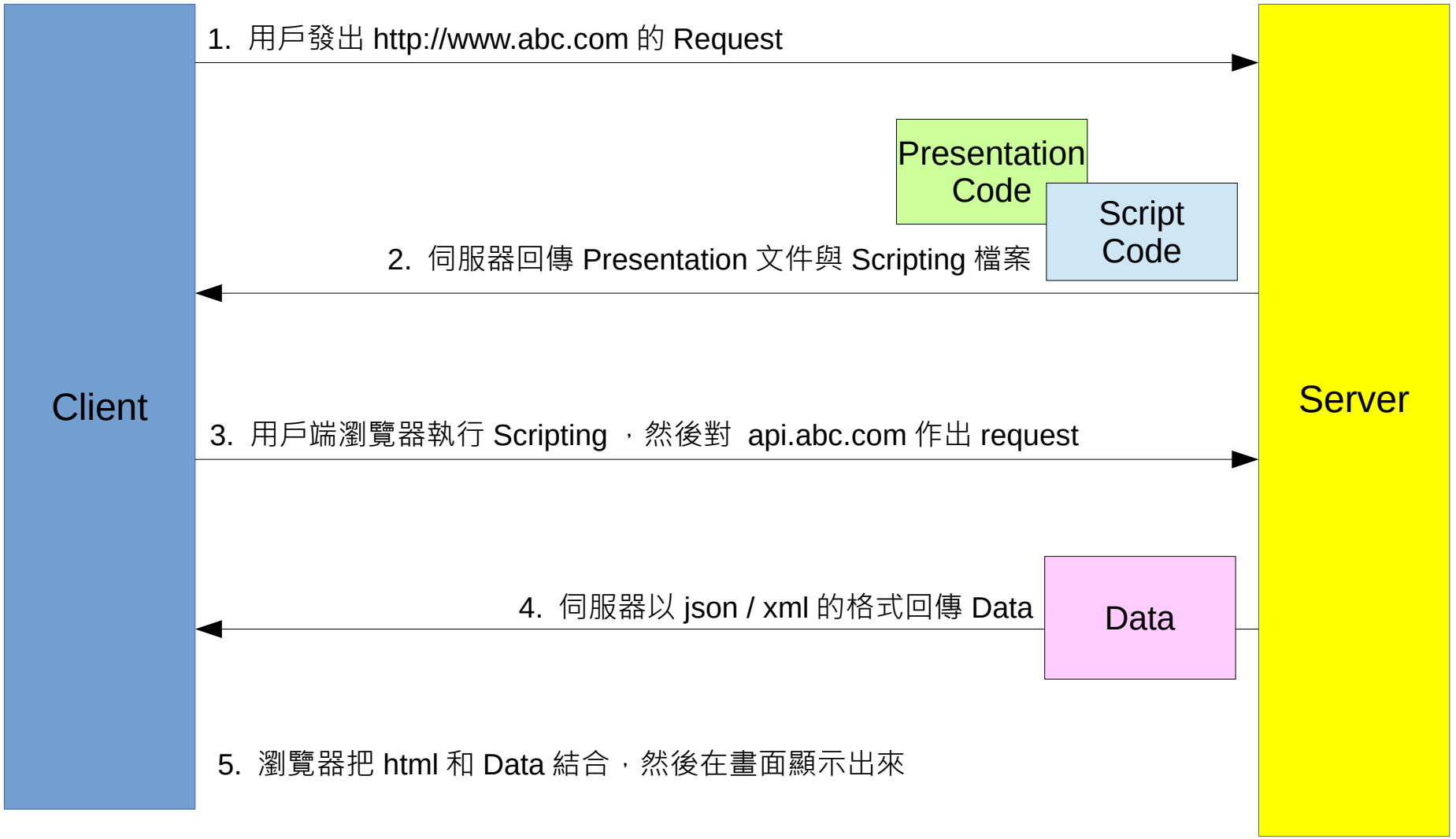
傳統網頁流程


Client
1. 用戶發出 http://www.abc.com 的 Request
2. 伺服器根據準備 Data 和 Presention logic
Presentation Code
Data
3. 伺服器把 Data 和 Presentation 結合成完整的 html
Result html
4. 伺服器把 html 丟回用戶
Server

可怕的程式碼與數據交錯

- Data 和 Presentation 混合， Presentation 無法單獨 cache ，只要 Data 變動就必須重新產生
- 無法徹底分割 Presentation 、 Business Logic 、和 Data ，影響開發、測試、除錯的協同作業
- 伺服器負責將 Data 和 Presentation 整合（簡稱 Data Binding ），消耗伺服器的 CPU 資源

前端後端分離架構


Client
1. 用戶發出 http://www.abc.com 的 Request
Presentation Code
Script Code
2. 伺服器回傳 Presentation 文件與 Scripting 檔案
Server
3. 用戶端瀏覽器執行 Scripting ，然後對 api.abc.com 作出 request
4. 伺服器以 json / xml 的格式回傳 Data
Data
5. 瀏覽器把 html 和 Data 結合，然後在畫面顯示出來

優點 1 ：客戶端快取節省流量

- 表面上看起來，原先僅需一次 request 增加為兩次，似乎花了更多的時間與網路流量！
- 但是，只要負責美工的同事不改變網頁外表。第一個 request 伺服器會回答 HTTP 304 Not Modified
- 因為網頁的 Presentation 不用每次重傳，可以節省不少網路流量（ = 省下錢 $ ）
  - 延伸思考： CDN

優點 2 ：資料與介面分離

- 現在 Presentation 和 Data 已經徹底分離！
- 開發和測試 Presentation 的 data mocking 可以使用專門的 mock API 伺服器。
- 只要準備好 API 與測試資料，前端工程師就能專心工作。前端開發延誤和後端工程師無關，不必背黑鍋 ~

優點 3 ：節省伺服器計算資源

- 可以不在伺服器做的東西，便不要放在伺服器
- Data Binding 現在是交給瀏覽器處理，不再是伺服器的工作，省下伺服器的 CPU 計算資源！

優點４：能重用的伺服器程式

- 即使未來要支持 IOS ， Android ，只要背後邏輯不變，這些 API 將能全部重覆使用，節省開發時間！

RESTful API 設計

RESTful 序言

- RESTful 只是回歸最初 HTTP 時代的設計哲學，不是什麼新東西
- 雖然大部份人（全部？）都是使用 HTTP 來實現 RESTful ，但是 RESTful 是設計哲學，不強制用什麼方式來實現

先談 RESTful 謬誤

- server side 需要 stateless
  - 正解 1 ： HTTP 只要求 protocol 本身是 stateless ，從未要求伺服器是 Stateless
  - 正解 2 ： RESTful 只要求 Application Server 是 stateless ， Server Side 其他部份可以 stateful
- 一定需要用上 HTTP 的 GET, PUT/PATCH, POST, DELETE
  - 正解：統一介面 (Uniform interface) 只建議使用 HTTP ，使用其他技術實作的統一介面也能滿足 RESTful 要求
  - 正解：沒人反對過你自建 Readonly endpoint ，別問 GraphQL 怎配 RESTful 了

RESTful 要求

- Client-server（客戶端和伺服器結構）
- Cacheable（能夠利用快取機制增進性能）
- Layered system（分層式系統）
- Stateless protocol（通訊協定具有無狀態性）
- Uniform interface（統一介面）

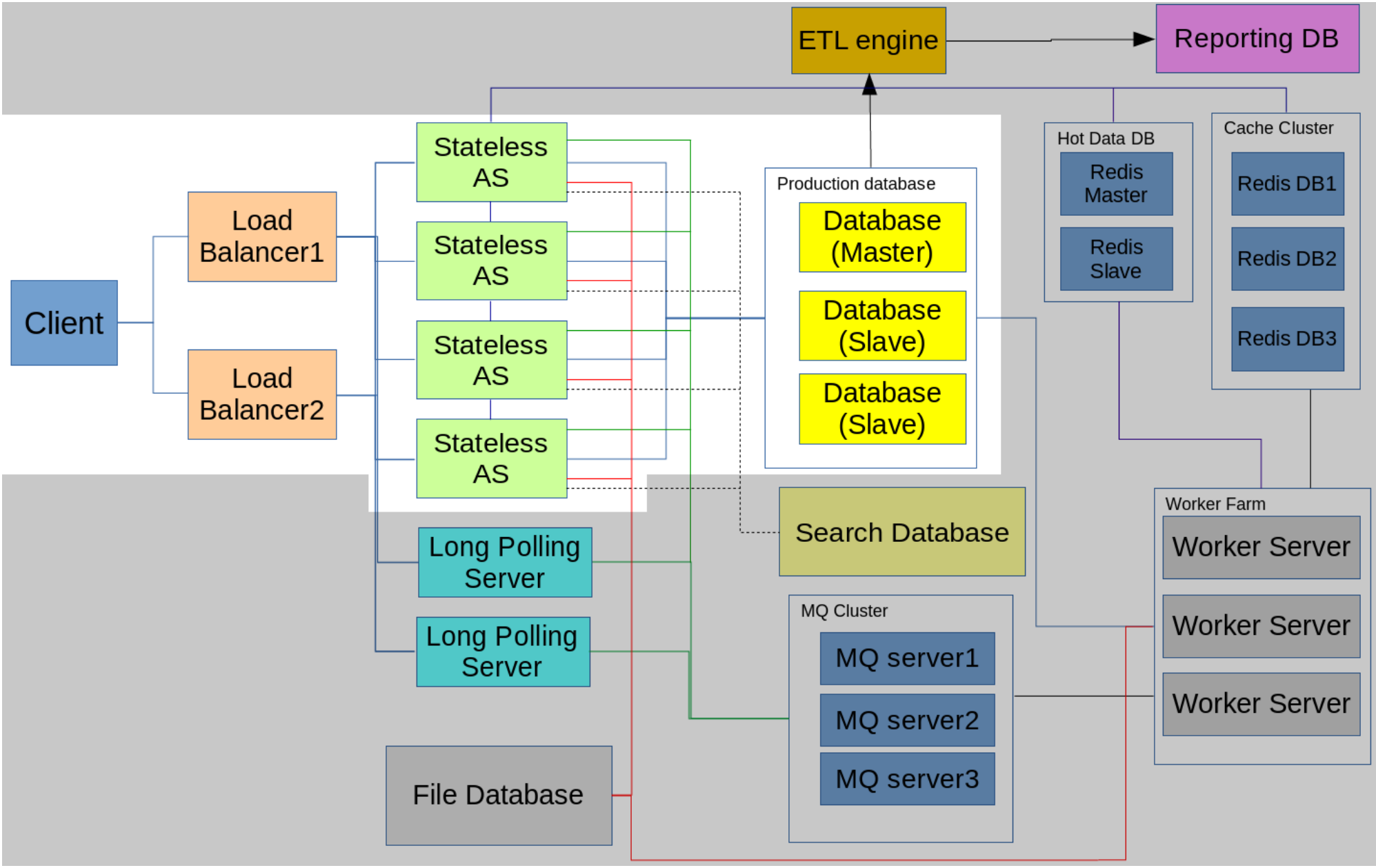
客戶端和伺服器結構

- 滿足 RESTful 的系統必需存在客戶端和伺服器端
- RESTful 只規範伺服器，和客戶端與伺服器端之間的通訊協定
  - 就是說：客戶端之間的 P2P 通訊，不在 RESTful 規範內

能夠利用快取機制增進性能

- 系統內的所有東西都必須定義能否使用快取
- 客戶端可以建立本地快取
- 在下次請求時，客戶會把本地快取版本告訴伺服器。如果伺服器發現資源沒有改動，反回 HTTP 304
- 可以省下伺服器 CPU 資源和網路流量（ = 省錢）
- 不常變動的物件適合設定為可快取，如網頁界面
- 詳見： HTTP conditional GET

分層式系統


ETL engine
Reporting DB
Client
Load Balancer1
Load Balancer2
Stateless AS
Stateless AS
Stateless AS
Stateless AS
Production database
Database (Master)
Database (Slave)
Database (Slave)
Hot Data DB
Redis Master
Redis Slave
Cache Cluster
Redis DB1
Redis DB2
Redis DB3
Search Database
Long Polling Server
Long Polling Server
MQ Cluster
MQ server1
MQ server2
MQ server3
Worker Farm
Worker Server
Worker Server
Worker Server
File Database

以下內容非常重要，直接影響到閣下未來爆肝度
睡著了的朋友請起來聽講
（謎之聲：睡了的朋友又怎能看到這段文字？）

淺談網路的不穩定性

- 隨時會斷線！往往看起來正常，其實已經斷線
- 使用 API 時，連線可能在信號送達伺服器前斷線
- 也可能在伺服器正要傳回 API 結果時斷線
- 更可能客戶端以為斷線，卻早已成功傳送資訊
- 除非使用同一個 TCP/IP 連線，否則當客戶端先發出 < 指令 1 > ，然後再發出 < 指令 2 > ，在次序錯亂的情況下，伺服器有可能先收到 < 指令 2 >
- 最簡單來說：任何可以錯的東西，都有機會出錯

無狀態通訊協定

- 通訊協定具有無狀態性
- atomic（原子性）
- 足夠完整的資料
- Idempotence（重新呼叫）

通訊協定具有無狀態性

- RESTful 伺服器端是被允許有狀態的
- Application Server 和 Connection 必須是無狀態
- RESTful 准許把狀態儲存到資料庫中，例如將登入狀態的短期資訊放到短期資料庫 (Redis)
- 最簡單的程式說法： RESTful 禁止使用 HttpSession(Java) ， session_start(PHP)

Stateless Application Server

- 你可以使用 HttpSession ，但是其資料必須放在 global storage ，而不是 local memory
- global storage 例子：
  - Redis
  - NFS（別亂來啊）
  -
  -
- 本來就是因為 HTTP 本身是 stateless protocol 才發明的嘛
-
- 在 RESTful 世界下，應該直接以（商品 A ，商品 B ）作結算，而不使用 HttpSession 去暫存

Stateless connection

- 你的 Request 可以引用 server side 上的資料
- 但是那一份資料必需要是 immutable 的
- non-RESTful 的購物車作例子：
  - 1. 加入商品 A 到購物車中
  - 2. 加入商品 B 到購物車中
  - 3. 結算
- 如果第二步的請求比第三步的請求更早到達伺服器，那用戶便不會購買商品 B
- 因為系統最終狀態是基於請求的到達是否合乎順序，所以這樣的通訊便是有狀態性

Stateless connection（續）

- RESTful 的購物車：
  - 加入商品 A 到購物車中
    - 伺服器回傳 ok ，現在購物車版本為 v1
  - 加入商品 B 到購物車中
    - 伺服器回傳 ok ，現在購物車版本為 v2
- 對 v2 版本的購物車結算

atomic（原子性）

- atomic 是由商業邏輯定義的
  - 從用戶 A 轉帳到用戶 B
  - 付錢買火車票
- 每一個動作都必需是 atomic 。一個完整執行後的 API 呼叫不能讓伺服器端數據停留在不一致的狀態。
- Each action call should be atomic. Any Successful API call should not leave server side in inconsistent state.
- 另一個說法：你不能使用二個（或以上） API 呼叫去完成一個「動作」

不符合 atomic 的範例

- 剛才用戶 A 轉帳到用戶 B ，如果是使用 2 個步驟：
  - /v1/users/A (PUT)
  - /v1/users/B (PUT)
- 如果從 A 扣了錢，然後客戶端 ~~internet explorer~~ 當掉
  - ~~改用 firefox 便好？~~
  - 結果是：系統會從用戶 A 扣錢了，卻沒把錢加到用錢 B 上

Non-atomic 下的前端土法煉鋼

- 轉帳前先把這記錄存放在客戶端硬碟
  - 程式重新啟動時發現未完成的轉帳記錄時，客戶端需要：
  - 先問伺候器，轉帳過程是停在那一步驟
  - 從當掉的步驟重來
- 不過對於硬碟故障的客戶，這方案沒用！

Non-atomic 下的後端土法煉鋼

- 藉 POST_DATA 中的 description 作配對
  - 找出所有扣了 A 的錢而沒有為 B 加錢的轉帳，然後作出數據回復
  - 需要定期掃瞄資料庫中的所有記錄，引起嚴重效能問題
- 掃瞄程式超級難寫，後端工程師肝病可能性大升

符合 atomic 的範例

- 建立新的 RESTful API /v1/MoneyTransfer (POST)
- POST_DATA =

```
{
    "from_user_id": "A",
    "to_user_id": "B",
    "amount": 100,
    "create_time" : "2014-07-05T16:35:46"
}
```

- 當用戶當機或網路斷線，再重新執行一次 API 即可！

足夠完整的資料

- 每個 API 呼叫時都需要向伺服器提供足夠完整的資料
  Each API call should provide sufficient information for server side to process.
- 你不應該假設伺服器知道「現在狀態」
  - 錯誤例子：玩中國象棋時，單純把用戶端指令用「炮二平五」來回傳給伺服器

不完整的資料範例

- 在紅方第 31 步行動時，紅方發出「炮二平五」指令
- 這時網絡某個路由器發生嚴重錯亂，把「炮二平五」的網絡封包送到火星
- 紅方電腦等 10 秒後沒有收到伺服器回應，以為封包丟失，於是再發出「炮二平五」指令，這個封包經由正常的路由器，所以 1 秒內就被傳送到伺服器
- 10 分鐘後，送往火星的錯誤封包終於被送到伺服器
- 這時已經到達第 55 步紅方行動，剛好紅方有炮棋在二線，伺服器沒法知道這是第 31 步時的封包，並且以為這是第 55 步的指令，因此執行「炮二平五」的指令，並且拒絕之後用戶真正第 55 步的指令

「足夠完整」的資料（續１）

- RESTful 的「足夠資料」，在中國象棋中最簡單實現的方法便是把每隻棋子的位置以及棋局的最後更改時間，在「炮二平五」的指令中同時傳給伺服器。
- 這樣伺服器便有足夠資料去分辨第 31 步時路經火星的封包。（因為棋局狀況和最後更改時間不符合）

「足夠完整」的資料（續２）

- 每次都把完整狀態丟來丟去是白痴，因為這將會大量消耗網絡封包
- 如果這是梭哈遊戲，你有機會可以攔截封包，事先得知對手蓋牌
- 所以：基於網路效能和資安考量， API 所用的資料可以放在伺服器， API 只需說明正在引用那份資料
  - 再說一次：只要你引用的資料是 immutable ，就不會有問題

「完整資料」範例

/chinese_chess_game/{game_id} (PUT)

```
{
    "step_id" : 31,
    "action" : " 炮二平五 "
}
```

- 伺服器收到這個 Request 時，會從短期資料庫中拿出第 31 步的棋局。
- 如果第 31 步已經被執行，伺服器會檢查已經存在的第 31 步是否等於 " 炮二平五 " 。如果是，傳回 " 成功 " 給客戶端；如果不是，則傳回錯誤。

Idempotence（重新呼叫）

- 在客戶端，任何未成功收到傳回訊息的 API ，只需要重新呼叫。
- 伺服器端需要懂得處理重覆 API request
- 在建立新物件時，把客戶端時間也放到 API request 中，讓伺服器能分辨是否收到重覆的 HTTP POST

統一介面

- 所有 URL 都應該基於物件，而不是行動
- 一個物件正常有 4 種行動：查詢，更改，誕生，刪除
  - 查詢：一般使用 HTTP GET method
  - 更改：一般使用 HTTP PUT/PATCH method
  - 誕生：一般使用 HTTP POST method
  - 刪除：一般使用 HTTP DELETE method

統一介面範例

- /v3/user/1234/book/54389
  - v3 = 使用第三版本的 /user/1234/book/54389 API
  - user/1234/book/54389 = 這是關於使用者 1234 屬下的書本 54389
    - GET = 取得書本內容
    - PUT = 更改書本內容
    - DELETE = 刪除書本紀錄
- /v3/user/1234/book/
  - GET = 取得使用者 1234 屬下的書本
  - POST = 在使用者 1234 屬下建立新的書本

完

謎語

為何少林寺十八銅人陣那麼少人能通過考驗？