序

1997 年，我讀高三的那一年，和朋友用學校圖書館的電腦上網，當時他告訴我，只要按一下檢視→原始碼就可以查看網頁的基礎程式碼。幾天後，另一個朋友示範給我看如何發佈自己的 HTML。我大開眼界。

在那之後，我就深深著迷。我到處瀏覽自己喜歡的網站並從中學習，拼湊出我自己的網站。我把大部分的空閒時間都花在家中餐廳的組裝電腦上，不斷摸索。我甚至「編寫」（其實只是複製貼上）我的第一個 JavaScript 對連結套用懸停樣式，這在當時還無法用簡單的 CSS 辦到。

經過一連串轉折後，我自己架設的音樂網站獲得相當高的知名度，我覺得這簡直就是電影《成名在望》的另類宅男版。因此，我收到郵寄的宣傳 CD，並且被列入音樂會的嘉賓名單。但對我來說，更重要的是我能與世界各地的人分享我的興趣。我當時是個住在郊區的無聊青少年，熱愛音樂，卻能接觸我完全不認識的人。這在當時帶給我很大的成就感，現在仍是如此。

我們現在只要用網頁技術就能建構強大的應用程式，但一開始可能很艱難。API 是提供資料的隱形背景、檢視→原始碼可顯示串連和縮小的程式碼、驗證和安全性深奧難懂，將這些東西放在一起可能令人不知所措。如果我們能夠看透這些令人困惑的細節，會發現我在 20 多年前摸索的技術現在可用來建構強大的網頁應用程式、編寫原生行動應用程式、建立強大的桌面應用程式、設計 3D 動畫，甚至編寫機器人程式。

碁峰 www.gotop.com.tw

身為教育工作者，我發現對許多人來說，最有效的學習方式是建構新事物、加以分解並根據自己的使用案例進行調整。這就是本書的目標。如果您瞭解一些 HTML、CSS 和 JavaScript，但不確定如何利用這些元件建構您想像中的穩健應用程式，這本書就很適合您。我將引導您建構 API，驅動網頁應用程式、原生行動應用程式和桌面應用程式的使用者介面。最重要的是，您將瞭解這些元件如何結合，進而創造美好的事物。

我等不及想看您的作品。

— *Adam*

碁峰 www.gotop.com.tw

前言

我在寫完第一個 Electron 桌面應用程式後，腦海中浮現了寫這本書的想法。從事網頁開發工作後，我立刻被使用網頁技術建構跨平台應用程式的可能性給迷住了。同時，React、React Native 和 GraphQL 都逐漸起飛。我尋找資源來瞭解這些東西是如何結合，但始終一無所獲。這本書就是我當時需要的指南。

本書的終極目標是介紹使用單一程式設計語言 JavaScript 建構各種應用程式的可能性。

目標讀者

本書適合有一些 HTML、CSS 和 JavaScript 經驗的中階開發人員，或希望學習經營事業或副業所需工具的初學者。

本書結構

本書旨在引導您開發適用於各種平台的範例應用程式，共分為以下章節：

- 第 1 章引導您建立 JavaScript 開發環境。
- 第 2–10 章介紹如何使用 Node、Express、MongoDB 和 Apollo Server 建構 API。
- 第 11–25 章探討使用 React、Apollo 及各種工具建構跨平台使用者介面的細節。其中：
  - 第 11 章介紹使用者介面開發和 React。
  - 第 12–17 章示範如何使用 React、Apollo Client 和 CSS-in-JS 建構網頁應用程式。

碁峰 www.gotop.com.tw

— 第 18–20 章引導您建構簡單的 Electron 應用程式。
— 第 21–25 章介紹如何使用 React Native 和 Expo 建構適用於 iOS 和 Android 的行動應用程式。

本書編排慣例

本書使用的排版慣例如下：

斜體字（*Italic*）

表示新術語、URL、電子郵件地址、檔案名稱和副檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式清單，以及在段落中指涉程式元素，例如變數或函式名稱、資料庫、資料類型、環境變數、陳述式和關鍵字。

定寬粗體字（**`Constant width bold`**）

表示使用者應逐字輸入的命令或其他文字。

定寬斜體字（*`Constant width italic`*）

表示應替換成使用者提供的值或經由上下文決定的值之文字。

此圖示表示提示或建議。

此圖示表示一般說明。

此圖示表示警告或注意事項。

碁峰 www.gotop.com.tw

第三章

使用 Node 和 Express 的網頁應用程式

建置 API 之前，我們將建構基本的伺服器端網頁應用程式以做為 API 後端的基礎。我們將使用 Express.js 框架（*https://expressjs.com*），它是「適用於 Node.js 的極簡網頁框架」，這表示它的功能不多，但可配置性極高。我們將使用 Express.js 做為 API 伺服器的基礎，但 Express 也可用來建構功能齊全的伺服器端網頁應用程式。

網站和行動應用程式等使用者介面會在必須存取資料時與網頁伺服器通訊。這些可能是任何資料，從在網頁瀏覽器中轉譯頁面所需的 HTML 到使用者的搜尋結果不等。用戶端介面使用 HTTP 與伺服器通訊、資料要求透過 HTTP 從用戶端傳送到在伺服器上執行的網頁應用程式，而網頁應用程式隨後處理要求並將資料回傳至用戶端，同樣也是透過 HTTP。

在本章中，我們將建構一個小型伺服器端網頁應用程式，以做為 API 的基礎。我們將使用 Express.js 框架建構傳送基本要求的簡單網頁應用程式。

Hello World

既然您已瞭解伺服器端網頁應用程式的基礎知識，我們現在就開始吧！在 API 專案的 *src* 目錄中，建立名稱為 *index.js* 的檔案並加入：

```
const express = require('express');
const app = express();
```


碁峰 www.gotop.com.tw

```
app.get('/', (req, res) => res.send('Hello World'));

app.listen(4000, () => console.log('Listening on port 4000!'));
```

在此例中，首先我們需要 `express` 相依性，並使用匯入的 Express.js 模組建立 `app` 物件。然後，使用 `app` 物件的 `get` 方法指示應用程式在使用者存取根 URL（`/`）時傳送「Hello World」回應。最後，我們指示應用程式在連接埠 4000 上執行。如此一來，即可在 URL *http://localhost:4000* 從本機檢視應用程式。

接著要執行應用程式，請在終端機中輸入 **`node src/index.js`**。之後，應在終端機中看到顯示 `Listening on port 4000!` 的紀錄。在此情況下，您應能開啟瀏覽器視窗前往 *http://localhost:4000* 並看到圖 3-1 中的結果。

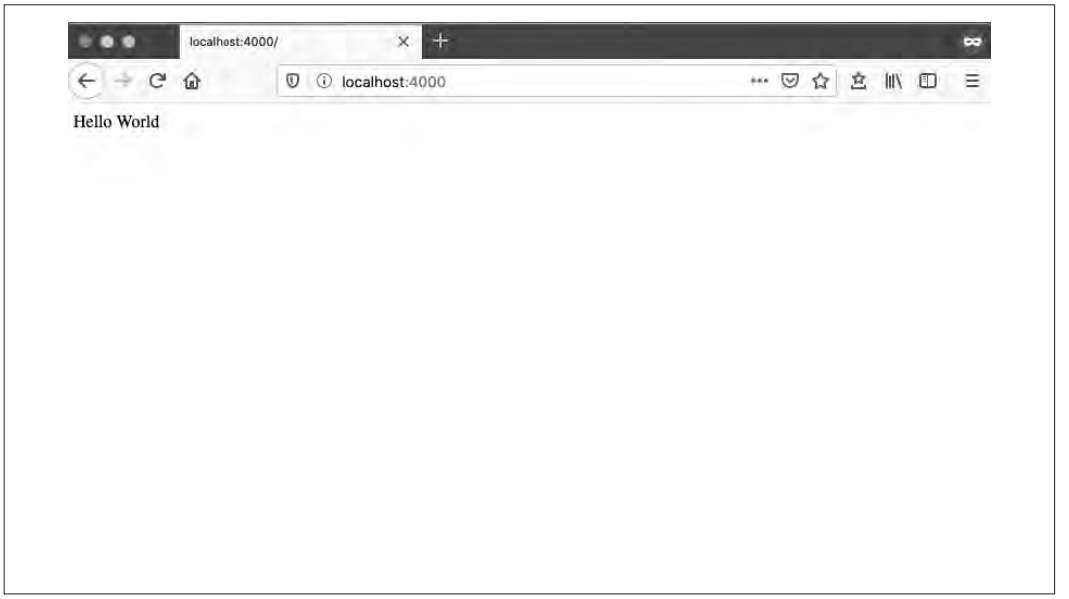

圖 3-1　瀏覽器中的 Hello World 伺服器程式碼結果

Nodemon

現在，假設此例的輸出未充分表達我們的興奮之情。我們要變更程式碼，在回應中加入驚嘆號。直接將 `res.send` 值改為顯示 `Hello World!!!`。完整的程式碼現在應該是：

```
app.get('/', (req, res) => res.send('Hello World!!!'));
```


碁峰 www.gotop.com.tw

如果移到網頁瀏覽器並重新整理頁面，您會發現輸出沒有改變。這是因為只要對網頁伺服器做任何變更，都必須重新啟動。因此，請切換回終端機並按 Ctrl + C 停止伺服器。接著，再次輸入 **`node index.js`** 以重新啟動。現在，回到瀏覽器並重新整理頁面時，應可看到更新後的回應。

您可以想像，每次變更都要停止並重新啟動伺服器很快就會變得令人厭煩。幸好，我們可以使用 Node 套件 `nodemon` 在變更時自動重新啟動伺服器。如果查看專案的 *package.json* 檔案，您會在 `scripts` 物件中看到 `dev` 命令，該命令指示 `nodemon` 監視 *index.js* 檔案：

```
"scripts": {
  ...
  "dev": "nodemon src/index.js"
  ...
}
```

package.json 指令碼

`scripts` 物件中有一些其他輔助命令。我們將在之後的章節中探討。

現在要從終端機啟動應用程式，請輸入：

```
npm run dev
```

移到瀏覽器並重新整理頁面，您會發現一切不變。為了確認 `nodemon` 自動重新啟動伺服器，我們再次更新 `res.send` 值，使其顯示：

```
res.send('Hello Web Server!!!')
```

現在應能在瀏覽器中重新整理頁面並看到更新，而不必手動重新啟動伺服器。

延伸連接埠選項

目前，我們的應用程式在連接埠 4000 上執行。這非常適合本機開發，但部署應用程式時，我們需要靈活的將此設為其他埠號。我們現在採取以下步驟來進行更新。首先加入 `port` 變數：

```
const port = process.env.PORT || 4000;
```


碁峰 www.gotop.com.tw

此變更讓我們得以在 Node 環境中動態設定連接埠，但在未指定連接埠時切換回連接埠 4000。接著，調整 `app.listen` 程式碼以適應此變更，並使用範本常值來記錄正確的連接埠：

```
app.listen(port, () =>
  console.log(`Server running at http://localhost:${port}`)
);
```

最終程式碼現在應為：

```
const express = require('express');

const app = express();
const port = process.env.PORT || 4000;

app.get('/', (req, res) => res.send('Hello World!!!'));

app.listen(port, () =>
  console.log(`Server running at http://localhost:${port}`)
);
```

我們現在已瞭解讓網頁伺服器程式碼開始運作的基礎知識。若要測試一切是否正常，請確認主控台中沒有錯誤，並在 *http://localhost:4000* 重新載入網頁瀏覽器。

結論

伺服器端網頁應用程式是 API 開發的基礎。在本章中，我們使用 Express.js 框架建構了基本的網頁應用程式。開發以 Node 為基礎的網頁應用程式時，有各種框架和工具可供選擇。Express.js 是很好的選擇，因為它具備靈活性、社群支援以及做為專案的成熟度。在下一章中，我們要將網頁應用程式變成 API。

碁峰 www.gotop.com.tw

第四章

第一個 GraphQL API

如果您在閱讀本文，表示您是一個人。人有許多興趣和愛好，還有家人、朋友、熟人、同學和同事。人也有自己的社交關係、興趣和愛好。某些關係和興趣重疊，某些則未重疊。總而言之，每個人都有由生活中的人們構成的連通圖。

此類互連資料正是 GraphQL 最初要解決的 API 開發挑戰。透過編寫 GraphQL API，我們能夠有效率地連接資料，進而降低要求的複雜性和數量，同時確切地為用戶端提供所需資料。

這聽起來是否對註記應用程式來說有點過頭？也許吧，但您會發現，GraphQL JavaScript 生態系統提供的工具和技術都能實現和簡化各種類型的 API 開發。

在本章中，我們將使用 `apollo-server-express` 套件建構 GraphQL API。因此，我們將探討基本的 GraphQL 主題、編寫 GraphQL 結構描述、開發程式碼以解析結構描述函式，並使用 GraphQL Playground 使用者介面存取 API。

將伺服器變成 API（類似）

我們開始 API 開發，使用 `apollo-server-express` 套件將 Express 伺服器變成 GraphQL 伺服器。Apollo Server（*https://oreil.ly/1fNt3*）是一種開放原始碼 GraphQL 伺服器函式庫，與大量的 Node.js 伺服器框架相容，包括 Express、Connect、Hapi 和 Koa。它讓我們能夠從 Node.js 應用程式提供資料以做為 GraphQL API，並且提供實用工具，例如 GraphQL Playground，這是在開發中用於處理 API 的視覺輔助工具。

碁峰 www.gotop.com.tw

為了編寫 API，我們將修改在上一章中編寫的網頁應用程式碼。首先加入 apollo-server-express 套件。請在 *src/index.js* 檔案的最上方加入：

```
const { ApolloServer, gql } = require('apollo-server-express');
```

我們已匯入 apollo-server，接著要建立基本的 GraphQL 應用程式。GraphQL 應用程式由兩個主要元件組成：類型定義的結構描述和解析程式，後者解析對資料執行的查詢和變動。如果這聽起來像是廢話，沒關係。我們將建置「Hello World」API 回應，並在整個 API 開發過程中進一步探討這些 GraphQL 主題。

首先，建立基本的結構描述，將它儲存在稱為 type Defs 的變數中。此結構描述將描述名稱為 hello 的單一 Query，它將回傳一個字串：

```
// 使用 GraphQL 結構描述語言建立結構描述
const typeDefs = gql`
  type Query {
    hello: String
  }
`;
```

現在我們已建立結構描述，我們可以加入解析程式，將值回傳給使用者。這將是回傳字串「Hello world!」的簡單函式：

```
// 為結構描述欄位提供解析程式函式
const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};
```

最後，我們將整合 Apollo Server 以提供 GraphQL API。為此，我們將增加一些 Apollo Server 專用設定和中介軟體並更新 app.listen 程式碼：

```
// Apollo Server 設定
const server = new ApolloServer({ typeDefs, resolvers });

// 套用 Apollo GraphQL 中介軟體並將路徑設為 /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```


碁峰 www.gotop.com.tw

綜合以上所述，*src/index.js* 檔案現在應該像這樣：

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

// 在 .env 檔案中指定的連接埠或連接埠 4000 上執行伺服器
const port = process.env.PORT || 4000;

// 使用 GraphQL 的結構描述語言建構結構描述
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

// 為結構描述欄位提供解析程式函式
const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};

const app = express();

// Apollo Server 設定
const server = new ApolloServer({ typeDefs, resolvers });

// 套用 Apollo GraphQL 中介軟體並將路徑設為 /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

如果您讓 nodemon 程序保持執行狀態，則可直接前往瀏覽器；否則，必須在終端機應用程式中輸入 `npm run dev` 以啟動伺服器。然後前往 *http://localhost:4000/api*，您會在其中看到 GraphQL Playground（圖 4-1）。此網頁應用程式隨附於 Apollo Server，是使用 GraphQL 的一大好處。您可以從此處執行 GraphQL 查詢和變動並查看結果。您也可以按一下結構描述索引標籤以存取為 API 自動建立的文件。

碁峰 www.gotop.com.tw

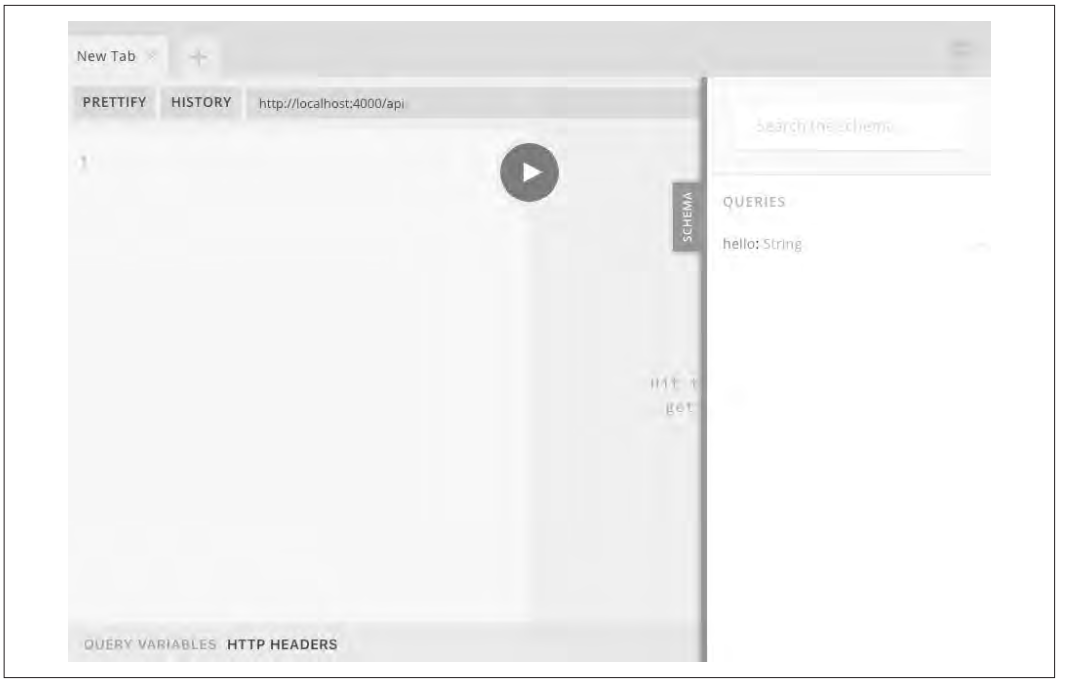

圖 4-1　GraphQL Playground

GraphQL Playground 採用深色的預設語法主題。在本書中，我將使用「淺色」主題以提高對比。請按一下齒輪圖示，在 GraphQL Playground 的設定中配置主題。

我們現在可以針對 GraphQL API 編寫查詢。為此，請在 GraphQL Playground 中輸入：

```
query {
  hello
}
```

按一下播放按鈕時，查詢應回傳以下內容（圖 4-2）：

```
{
  "data": {
    "hello": "Hello world!"
  }
}
```


碁峰 www.gotop.com.tw

圖 4-2　hello 查詢

好了！我們已透過 GraphQL Playground 存取有效的 GraphQL API。API 接受 `hello` 查詢並回傳字串 `Hello world!`。更重要的是，我們現在有了建構功能齊全的 API 所需的結構。

GraphQL 基礎知識

在上一節中，我們已探討並開發第一個 API，但讓我們花一些時間來回顧一下 GraphQL API 的不同部分。GraphQL API 的兩個主要構件是**結構描述**和**解析程式**。只要瞭解這兩個元件，就能更有效地將它們應用在 API 設計和開發。

結構描述

結構描述是資料和互動的書面表示法。透過要求結構描述，GraphQL 對 API 實施嚴格的計畫。這是因為 API 只能根據結構描述的定義來回傳資料和執行互動。

碁峰 www.gotop.com.tw

GraphQL 結構描述的基本元件是物件類型。在上一個例子中，我們建立了 GraphQL 物件類型 `Query`，欄位是 `hello`，回傳的純量類型是 `String`。GraphQL 包含五種內建純量類型：

`String`

採用 UTF-8 字元編碼的字串

`Boolean`

真假值

`Int`

32 位元整數

`Float`

浮點值

`ID`

唯一識別碼

透過這些基本元件，我們可以為 API 建立結構描述。首先定義類型。假設我們要為披薩菜單建立 API。我們可以定義 GraphQL 結構描述類型 `Pizza`，如下所示：

```
type Pizza {
}
```

每個披薩都有唯一 ID、大小（例如小、中、大）、片數和選用配料。`Pizza` 結構描述可能像這樣：

```
type Pizza {
  id: ID
  size: String
  slices: Int
  toppings: [String]
}
```

在此結構描述中，某些欄位值為必要值（例如 ID、大小、片數），其他則是選用值（例如配料）。我們可以使用驚嘆號來表示欄位必須包含值。我們更新結構描述以表示必要值：

碁峰 www.gotop.com.tw

```
type Pizza {
  id: ID!
  size: String!
  slices: Int!
  toppings: [String]
}
```

在本書中，我們將編寫基本結構描述，這將讓我們能夠執行常見 API 中絕大多數的操作。如果想要探索所有 GraphQL 結構描述選項，建議您閱讀 GraphQL 結構描述文件（*https://oreil.ly/DPT8C*）。

解析程式

GraphQL API 的第二個部分是解析程式。顧名思義，解析程式的用途是**解析** API 使用者要求的資料。我們將編寫解析程式，首先在結構描述中加以定義，然後在 JavaScript 程式碼中建置邏輯。我們的 API 將包含兩種解析程式：查詢和變動。

查詢

查詢以所需格式從 API 要求特定資料。在我們假設的披薩 API 中，我們可以編寫將回傳菜單上披薩完整清單的查詢，以及另一個將回傳單一披薩詳細資訊的查詢。查詢隨後將回傳物件，包含 API 使用者要求的資料。查詢決不會修改資料，只會存取資料。

變動

要修改 API 中的資料時，我們使用變動。在披薩例子中，我們可以編寫變更披薩配料的變動，以及另一個讓我們調整片數的變動。如同查詢，變動也將以物件形式回傳結果，通常是所執行操作的最終結果。

調整 API

您已充分瞭解 GraphQL 的元件，接著要為註記應用程式調整初始 API 程式碼。首先，我們將編寫一些程式碼來讀取和建立註記。

首先需要一些資料供 API 處理。我們建立「註記」物件陣列，以做為 API 提供的基本資料。隨著專案進展，我們會將此記憶體中的資料表示法換成資料庫。目前，我們將資料儲存在名稱為 `notes` 的變數中。陣列中的每個註記都是具有 `id`、`content`、`author` 三個屬性的物件：

碁峰 www.gotop.com.tw

```
let notes = [
  { id: '1', content: 'This is a note', author: 'Adam Scott' },
  { id: '2', content: 'This is another note', author: 'Harlow Everly' },
  { id: '3', content: 'Oh hey look, another note!', author: 'Riley Harrison' }
];
```

我們已取得一些資料，接著要調整 GraphQL API 以處理資料。首先來看看我們的結構描述。我們的結構描述是資料及其互動方式的 GraphQL 表示法。我們知道會有註記，這些註記將被查詢和變動。這些註記目前包含 ID、內容和作者欄位。我們在 `typeDefs` GraphQL 結構描述中建立對應的註記類型。這將在 API 中表示註記的屬性：

```
type Note {
  id: ID!
  content: String!
  author: String!
}
```

接著加入查詢，以便擷取所有註記的清單。我們更新 `Query` 類型以加入 `notes` 查詢，它將回傳註記物件的陣列：

```
type Query {
  hello: String!
  notes: [Note!]!
}
```

現在，我們可以更新解析程式碼以執行回傳資料陣列的工作。我們更新 `Query` 程式碼以加入以下 `notes` 解析程式，它將回傳原始資料物件：

```
Query: {
    hello: () => 'Hello world!',
    notes: () => notes
  },
```

如果現在前往在 *http://localhost:4000/api* 執行的 GraphQL Playground，就可以測試 `notes` 查詢。請輸入以下查詢：

```
query {
  notes {
    id
    content
    author
  }
}
```


碁峰 www.gotop.com.tw

隨後，按一下播放按鈕時，應看到回傳 `data` 物件，其中包含資料陣列（圖 4-3）。

圖 4-3　notes 查詢

GraphQL 最酷的一點是，我們可以移除我們要求的任何欄位，例如 `id` 或 `author`。此時，API 會確切地回傳我們要求的資料。這讓使用資料的用戶端得以控制在各個要求中傳送的資料量，並將該資料限制在所需範圍內（圖 4-4）。

碁峰 www.gotop.com.tw

圖 4-4　只要求 content 資料的 notes 查詢

我們現在可以查詢註記的完整清單，接著編寫一些程式碼，以便查詢單一註記。您可以從使用者介面的觀點想像其實用性，顯示包含單一特定註記的畫面。為此，我們必須要求具有特定 `id` 值的註記。我們必須在 GraphQL 結構描述中使用**引數**。引數允許 API 使用者將特定的值傳遞給解析程式函式，提供解析所需的資訊。我們來新增一個 `note` 查詢，它將使用 `id` 引數，類型為 `ID`。我們在 `typeDefs` 中更新 `Query` 物件，加入新的 `note` 查詢：

```
type Query {
  hello: String
  notes: [Note!]!
  note(id: ID!): Note!
}
```

更新結構描述後，我們可以編寫查詢解析程式以回傳要求的註記。為此，我們必須能夠讀取 API 使用者的引數值。幸好，Apollo Server 可傳遞以下實用參數給解析程式函式：

`parent`

父查詢的結果，在進行巢狀查詢時很實用。

碁峰 www.gotop.com.tw

`args`

這些是使用者在查詢中傳遞的參數。

`context`

從伺服器應用程式傳遞到解析程式函式的資訊。這可能包括目前使用者或資料庫資訊等等。

`info`

關於查詢本身的資訊。

我們將視需要在程式碼中進一步探索這些參數。如果感到好奇，您可以在 Apollo Server 文件（*https://oreil.ly/l6mL4*）中深入瞭解這些參數。目前，我們只需要第二個參數 `args` 中包含的資訊。

`note` 查詢將使用註記 `id` 做為引數並且在 `note` 物件陣列中尋找。請在查詢解析程式碼中加入：

```
note: (parent, args) => {
  return notes.find(note => note.id === args.id);
}
```

解析程式碼現在應如下所示：

```
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  }
};
```

為了執行查詢，我們回到網頁瀏覽器並前往 GraphQL Playground：*http://localhost:4000/api*。我們現在可以查詢具有特定 `id` 的註記，如下所示：

```
query {
  note(id: "1") {
    id
    content
    author
  }
}
```


碁峰 www.gotop.com.tw

執行此查詢時，得到的結果應是具有要求之 id 值的註記。如果嘗試查詢不存在的註記，應會得到值為 null 的結果。為了進行測試，請嘗試變更 id 值以回傳不同的結果。

讓我們導入使用 GraphQL 變動建立新註記的功能來完成初始 API 程式碼。在該變動中，使用者將傳遞註記的內容。現在，我們將對註記作者進行硬編碼。首先，用 Mutation 類型更新 typeDefs 結構描述，我們將稱之為 newNote：

```
type Mutation {
  newNote(content: String!): Note!
}
```

接著編寫變動解析程式，它將接收註記內容做為引數、將註記儲存為物件，並且在記憶體中將其新增至 notes 陣列。為此，我們將新增 Mutation 物件至解析程式。在 Mutation 物件中，我們將新增稱為 newNote 的函式，它包含了 parent 和 args 參數。在此函式中，我們將取得引數 content 並建立具有 id、content 和 author 機碼的物件。您可能已發現，這與目前的註記結構描述相符。我們隨後會將此物件推送至 notes 陣列並回傳物件。回傳物件允許 GraphQL 變動以預定格式接收回應。請編寫以下程式碼：

```
Mutation: {
  newNote: (parent, args) => {
    let noteValue = {
      id: String(notes.length + 1),
      content: args.content,
      author: 'Adam Scott'
    };
    notes.push(noteValue);
    return noteValue;
  }
}
```

src/index.js 檔案現在如下所示：

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

// 在 .env 檔案中指定的連接埠或連接埠 4000 上執行伺服器
const port = process.env.PORT || 4000;

let notes = [
  { id: '1', content: 'This is a note', author: 'Adam Scott' },
  { id: '2', content: 'This is another note', author: 'Harlow Everly' },
  { id: '3', content: 'Oh hey look, another note!', author: 'Riley Harrison' }
];
```


碁峰 www.gotop.com.tw

```
// 使用 GraphQL 的結構描述語言建構結構描述
const typeDefs = gql`
  type Note {
    id: ID!
    content: String!
    author: String!
  }

  type Query {
    hello: String
    notes: [Note!]!
    note(id: ID!): Note!
  }

  type Mutation {
    newNote(content: String!): Note!
  }
`;

// 為結構描述欄位提供解析程式函式
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  },
  Mutation: {
    newNote: (parent, args) => {
      let noteValue = {
        id: String(notes.length + 1),
        content: args.content,
        author: 'Adam Scott'
      };
      notes.push(noteValue);
      return noteValue;
    }
  }
};

const app = express();

// Apollo Server 設定
const server = new ApolloServer({ typeDefs, resolvers });
```


碁峰 www.gotop.com.tw

```
// 套用 Apollo GraphQL 中介軟體並將路徑設為 /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

更新結構描述和解析程式以接受變動後，我們在 GraphQL Playground 中測試看看：*http://localhost:4000/api*。在 Playground 中，按一下 + 符號建立新的索引標籤並編寫變動：

```
mutation {
  newNote (content: "This is a mutant note!") {
   content
   id
   author
  }
}
```

按一下播放按鈕時，會得到包含新註記的內容、ID 和作者的回應。您也可以重新執行 `notes` 查詢以檢查變動是否有效。為此，請切換回包含該查詢的 GraphQL Playground 索引標籤，或輸入：

```
query {
  notes {
    content
    id
    author
  }
}
```

此查詢執行時，應看到四個註記，包括最近新增的註記。

資料儲存

我們目前將資料儲存在記憶體中。因此，每當重新啟動伺服器，就會遺失資料。在下一章中，我們將使用資料庫來保存資料。

我們已成功建置查詢和變動解析程式並且在 GraphQL Playground 使用者介面中進行測試。

碁峰 www.gotop.com.tw

結論

在本章中，我們使用 `apollo-server-express` 模組成功建構了 GraphQL API。我們現在可以對記憶體中資料物件執行查詢和變動。此設定提供用來建構任何 API 的穩固基礎。在下一章中，我們將探討使用資料庫來保存資料的能力。

碁峰 www.gotop.com.tw