第十五章

REST API 與 JSON

到目前為止，我們已經設計出一個可在瀏覽器顯示的網站了。現在我們將焦點轉移到：讓資料與功能可供其他程式使用。Internet 再也不是孤立網站的群集了，愈來愈像真正的 Web：網站可自由地互相通訊，為使用者提供更豐富的體驗。這對程式員而言是美夢成真的一刻：Internet 已經可以像真人一樣操作你的程式。

在本章中，我們會在 app 加入一個 Web 服務（Web 伺服器與 Web 服務沒有理由不能待在同一個應用程式）。“Web 服務”這個名詞是一個通用名詞，代表所有可以透過 HTTP 操作的應用程式編程介面（API）。Web 服務的概念已經出現一段時間了，但一直到最近，這個技術仍然很古板、難懂且過度複雜。目前還有一些系統在使用這些技術（例如 SOAP 與 WSDL），而且也有 Node 套件可協助你與這些系統連接。但我們不會討論它，而是將焦點放在提供所謂的“RESTful”服務，這容易連接多了。

REST 縮寫代表“representational state transfer（含狀態傳輸）”，而令人難以理解的“RESTful”代表滿足 REST 原則的 Web 服務。REST 的正式定義很複雜，但 REST 基本上是一個介於用戶端與伺服器之間的無狀態連結。REST 的正式定義也指定服務要能被緩存，而且該服務可以層次化（也就是說，當你使用 REST API 時，在它下面可能有其他的 REST API）。

從實際的角度來看，HTTP 的限制確實讓人難以建立非 RESTful 的 API，舉例而言，這就像你必須走出自己的路才能建立自己的國家。所以我們大多可以勝任工作。

我們會在 Meadowlark Travel 網站添加 REST API。為了促銷 Oregon 旅遊，Meadowlark Travel 有一個景點資料庫，裡面有有趣的史實。有一個 API 可用來建構 app，讓訪客可以用他們的手機或平板來自由旅行：如果設備可以感知位置，app 可以讓他們知道是否

位於景點附近。所以這個資料庫會成長，API 也支援添加地標及景點（它會進入審查佇列，以避免濫用）。

JSON 與 XML

提供 API 的目的，在於讓你有一個共用的語言可以溝通。通訊的一部分跟我們有關：我們必須使用 HTTP 方法來與伺服器溝通。但除此之外，我們就可以自由地使用任何想要用的資料語言。傳統上，XML 已成為受歡迎的選擇，而且它仍然是一種重要的標記語言。雖然 XML 並沒有特別複雜，但 Douglas Crockford 看到它有輕量化的空間，因而造成 JavaScript 物件標記法（JSON）的誕生。它除了非常 JavaScript 友善之外（但不代表它是專用的：它是可以讓任何語言輕鬆解析的格式），它的另一個優點是比 XML 更容易手動編寫。

在大部分的應用程式中，我比 XML 還喜歡 JSON：它有更佳的 JavaScript 支援，而且它是更簡單、更緊湊的格式。我建議你將焦點放在 JSON，只在既有的系統需要使用 XML 來與你的 app 溝通時，才提供 XML。

我們的 API

我們會在實作 API 之前，先做好規畫。我們想要下列的功能：

`GET /api/attractions`

抓取景點。將 `lat`、`lng` 與 `radius` 作為查詢字串參數，並回傳景點清單。

`GET /api/attraction/:id`

回傳景點 ID。

`POST /api/attraction`

在查詢內文中，取用 `lat`、`lng`、`name`、`description` 與 `email`。新加入的景點會進入審核佇列。

`PUT /api/attraction/:id`

更新既有的景點。取用景點 ID、`lat`、`lng`、`name`、`description` 與 `email`。更新動作會進入審核佇列。

`DEL /api/attraction/:id`

刪除一個景點。取用景點 ID、`email` 與 `reason`。刪除的動作會進入審核佇列。

我們有很多方法可以描述 API。在這裡，我們選擇使用 HTTP 方法與路徑的結合來區別 API 呼叫，以及查詢字串與內文參數的混合來傳遞資料。我們可以在完全相同的方法中使用不同的路徑（例如 */api/attractions/delete*）[1]。我們也可以用一致的方式將資料傳入。例如，我們可選擇用 URL 傳遞所有必要的資訊來取出參數，而不是使用查詢字串：`GET /api/attractions/:lat/:lng/:radius`。為了避免過長的 URL，我建議使用請求內文來傳遞大區塊的資料（例如景點說明）。

使用 POST 來建構某些東西，以及使用 PUT 來更新（或修改）某些東西，已經成為一種標準做法。但是這兩個字的英文含義並未區分這兩個動作，所以你可考慮用路徑來區分這兩種動作，以避免疑惑。

為了簡單起見，我們只會實作這三種功能：添加景點，取回景點，及列出景點。如果你下載本書的資源的話，就可以看到完整的資訊。

API 錯誤回報

HTTP API 的錯誤回報通常使用 HTTP 狀態碼來實作：如果請求回傳 200（OK），用戶端就會知道請求成功了。如果請求回傳 500（內部伺服器錯誤），請求就是失敗的。但是，在大部分的應用程式中，並非所有東西都可以（或應該）被粗略的分類為“成功”或“失敗”。例如，如果你使用 ID 來請求某個東西，但是 ID 不存在呢？這不代表伺服器錯誤，而是用戶端要求某個不存在的東西。一般情況下，錯誤可被分成下列的幾類：

災難性錯誤

造成伺服器不穩定或進入未知狀態的錯誤。通常這是未被處理的例外所產生的結果。要從災難性錯誤回復，最安全的方法是重新啟動伺服器。理想情況下，任何等待中的請求都會收到一個 500 回應碼，但如果失敗太嚴重的話，伺服器可能完全無法回應，請求就會逾時。

可回復的伺服器錯誤

可回復的錯誤不需要重新啟動伺服器，或任何其他英雄式行動。這種錯誤是因為伺服器發生意外的錯誤情況（例如，資料庫連結無法使用）。這個問題可能是暫時或永久的。500 回應碼適用於這種情況。

1 如果你的用户端不能使用不同的 HTTP 方法，見 *https://github.com/expressjs/method-override*，它可以讓你“偽裝”不同的 HTTP 方法。

用戶端錯誤

用戶端錯誤是用戶端犯了錯誤造成的，通常是缺少參數或參數不正確。這不適合使用 500 回應碼，畢竟伺服器並沒有問題。每一件事都正常地運作，只是用戶端沒有正確地使用 API。你有幾種選擇：你可以回應一個 200 狀態碼，並在回應內文中說明錯誤，或另外使用適當的 HTTP 狀態碼，試著說明錯誤。我建議第二種做法。在這個情況下，最好用的回應碼是 404（未發現）、400（錯誤的請求）及 401（未經授權）。此外，回應內文應明確解釋錯誤。如果你想要向上提升，錯誤訊息甚至可以含有文件的連結。注意，如果使用者請求一串的東西，而且沒有東西回傳，這不是一種錯誤狀況，較適當的做法是回傳一個空的清單。

在我們的應用程式中，會在內文中使用 HTTP 回應碼與錯誤訊息的結合。注意，這種方法與 jQuery 相容，對於目前盛行使用 jQuery 來操作 API 的情況，這是很重要的考量要素。

跨來源資源共享（CORS）

如果你要發布 API，可能想要讓其他人也可以使用 API。這會產生**跨站** *HTTP* **請求**。跨站 HTTP 請求一直是受到攻擊的對象，因此被**同源政策**限制，它會限制何處的指令碼可被載入。具體來說，協定、網域及連接埠都必須符合。這可以讓其他的網站使用你的 API，這就是 CORS 的源起。CORS 可讓你根據各種情況來解除限制，甚至讓你具體列出哪些網域可以存取該指令碼。CORS 是透過 `Access-Control-Allow-Origin` 標頭來實作的。要在 Express 應用程式中實作它，最簡單的方式是使用 `cors` 套件（`npm install--save cors`）。要在你的應用程式中使用 CORS：

```
app.use(require('cors')());
```

會出現同樣來源的 API 是有原因的（為了避免攻擊），我建議你在必要時才套用 CORS。在我們的案例中，我們想要公開整個 API（但只有 API），所以我們要將 CORS 的路徑限制為以 */api* 開頭：

```
app.use('/api', require('cors')());
```

要知道更多進階 CORS 用法，請參考套件文件（*https://www.npmjs.org/package/cors*）。

我們的資料存儲

同樣的，我們會使用 Mongoose 建立一個資料庫中景點模型的架構。建立檔案 *models/attraction.js*：

```
var mongoose = require('mongoose');

var attractionSchema = mongoose.Schema({
    name: String,
    description: String,
    location: { lat: Number, lng: Number },
    history: {
        event: String,
        notes: String,
        email: String,
        date: Date,
    },
    updateId: String,
    approved: Boolean,
});
var Attraction = mongoose.model('Attraction', attractionSchema);
module.exports = Attraction;
```

因為我們想要審核更新，所以不能讓 API 隨便更新原始記錄。我們的方法是建立一筆參考原始記錄的新記錄（在它的 `updateId` 特性）。當記錄被審核之後，我們可以將原始記錄更新為更新記錄裡面的資訊，接著刪除更新記錄。

我們的測試

如果我們使用 HTTP 動詞而不是 `GET`，對於測試我們的 API 而言，將是件麻煩事，因為瀏覽器只知道如何發出 `GET` 請求（及表單的 `POST` 請求）。有一些方法可以處理這種情況，例如很棒的“Postman - REST Client” Chrome 外掛。但是無論你是否使用這種公用程式，最好還是要有自動化的測試機制。在我們為 API 編寫測試之前，需要一種方式來實際地呼叫 REST API。為此，我們會使用一種 Node 套件，稱為 `restler`：

```
npm install --save-dev restler
```

我們會將之後要實作的 API 回呼的測試項放在 *qa/tests-api.js* 裡面：

```
var assert = require('chai').assert;
var http = require('http');
var rest = require('restler');
```

```
suite('API tests', function(){

    var attraction = {
        lat: 45.516011,
        lng: -122.682062,
        name: 'Portland Art Museum',
        description: 'Founded in 1892, the Portland Art Museum\'s colleciton ' +
            'of native art is not to be missed.  If modern art is more to your ' +
            'liking, there are six stories of modern art for your enjoyment.',
        email: 'test@meadowlarktravel.com',
    };

    var base = 'http://localhost:3000';

    test('should be able to add an attraction', function(done){
        rest.post(base+'/api/attraction', {data:attraction}).on('success',
                function(data){
            assert.match(data.id, /\w/, 'id must be set');
            done();
        });
    });

    test('should be able to retrieve an attraction', function(done){
        rest.post(base+'/api/attraction', {data:attraction}).on('success',
                function(data){
            rest.get(base+'/api/attraction/'+data.id).on('success',
                    function(data){
                assert(data.name===attraction.name);
                assert(data.description===attraction.description);
                done();
            })
        })
    });

});
```

注意，在取回景點的測試中，我們先加入一個景點。或許你會想，我們並不需要做這件事，因為第一次測試已經做過了，但這個動作有兩個原因。第一個原因很實際：就算測試在檔案中看起來是按照順序，但是因為 JavaScript 的非同步性質，API 呼叫並不保證會按照順序來執行。第二個原因與原則有關：任何測試項必須完全獨立，不能依賴其他的測試項。

其語法必須簡單名瞭：我們呼叫 `rest.get` 或 `rest.put`，將 URL 以及一個含有 `data` 特性的選項物件傳給它，這個物件會被用來作為請求內文。方法會回傳一個發出事件的應

允。我們關心的是 success 事件。當你在應用程式中使用 restler 時，可能也想要監聽其他事件，如 fail （伺服器會回應 4xx 狀態碼）或 error（連結或解析錯誤）。進一步資訊請參考 restler 文件（*https://github.com/danwrong/restler*）。

使用 Express 來提供 API

Express 非常適合提供 API。在本章的稍後，我們會學習如何使用 Node 模組來做這件事，它也提供一些額外的功能，但我們會從一個單純的 Express 實作開始：

```
var Attraction = require('./models/attraction.js');

app.get('/api/attractions', function(req, res){
    Attraction.find({ approved: true }, function(err, attractions){
        if(err) return res.send(500, 'Error occurred: database error.');
        res.json(attractions.map(function(a){
            return {
                name: a.name,
                id: a._id,
                description: a.description,
                location: a.location,
            }
        }));
    });
});

app.post('/api/attraction', function(req, res){
    var a = new Attraction({
        name: req.body.name,
        description: req.body.description,
        location: { lat: req.body.lat, lng: req.body.lng },
        history: {
            event: 'created',
            email: req.body.email,
            date: new Date(),
        },
        approved: false,
    });
    a.save(function(err, a){
        if(err) return res.send(500, 'Error occurred: database error.');
        res.json({ id: a._id });
    });
});

app.get('/api/attraction/:id', function(req,res){
```

```
        Attraction.findById(req.params.id, function(err, a){
            if(err) return res.send(500, 'Error occurred: database error.');
            res.json({
                name: a.name,
                id: a._id,
                description: a.description,
                location: a.location,
            });
        });
    });
```

注意，當我們回傳一個景點的時候，不能只回傳資料庫所回傳的模型，那會公開內部的實作資訊。相反地，我們要挑選用來建構要回傳的新物件所需的資訊。

現在如果我們執行測試（使用 Grunt 或 `mocha -u tdd -R spec qa/tests-api.js`），我們會看到測試項已通過。

使用 REST 外掛程式

如同你所看到的，只要使用 Express 就可以編寫 API。但是使用 REST 外掛程式也有一些優點。我們來使用強健的 `connect-rest` 來對 API 做未來驗證。首先，安裝它：

```
npm install --save connect-rest
```

並將它匯入 *meadowlark.js*：

```
var rest = require('connect-rest');
```

我們的 API 不應該與一般的網站路由衝突（確保你沒有建立任何以 */api* 開頭的網站路由）。我建議在網站路由之後添加 API 路由：`connect-rest` 會檢視每一個請求並將特性添加到請求物件，並做額外的記錄。因此，它的最佳位置，是在你連結網路路由的後面，在 404 處理程式前面：

```
// 網站路由從這裡開始

// 在這裡使用 rest.VERB 定義 API 路由…

// API 配置
var apiOptions = {
    context: '/api',
    domain: require('domain').create(),
};
```

```
// 將 API 連結到管道
app.use(rest.rester(apiOptions));

// 404 處理程式從這裡開始
```

如果你想要盡量分隔網站與 API，考慮使用子網域，例如 *api.meadowlark.com*。稍後會看到它的範例。

connect-rest 已經給我們一些方便了：它可以讓我們自動地將所有的 API 呼叫前面加上 /api。這可減少打字錯誤的可能性，如果要的話，也可以讓我們輕鬆地改變基底 URL。

我們來看如何添加 API 方法：

```
rest.get('/attractions', function(req, content, cb){
    Attraction.find({ approved: true }, function(err, attractions){
        if(err) return cb({ error: 'Internal error.' });
        cb(null, attractions.map(function(a){
            return {
                name: a.name,
                description: a.description,
                location: a.location,
            };
        }));
    });
});

rest.post('/attraction', function(req, content, cb){
    var a = new Attraction({
        name: req.body.name,
        description: req.body.description,
        location: { lat: req.body.lat, lng: req.body.lng },
        history: {
            event: 'created',
            email: req.body.email,
            date: new Date(),
        },
        approved: false,
    });
    a.save(function(err, a){
        if(err) return cb({ error: 'Unable to add attraction.' });
        cb(null, { id: a._id });
    });
});
```

```
rest.get('/attraction/:id', function(req, content, cb){
    Attraction.findById(req.params.id, function(err, a){
        if(err) return cb({ error: 'Unable to retrieve attraction.' });
        cb(null, {
            name: attraction.name,
            description: attraction.description,
            location: attraction.location,
        });
    });
});
```

REST 函式並不是使用一般的請求 / 回應，而是使用三個參數：請求（與一般的一樣）、一個內容物件，它是被解析的請求內文、以及一個回呼函式，可在非同步 API 呼叫使用。因為我們正在使用資料庫，是非同步的，所以必須使用回呼來將回應傳送給用戶端（有一個同步 API，你可以參考 connect-rest 文件（*https://github.com/imrefazekas/connect-rest*））。

也請注意，當我們建立 API 時，會指定一個網域（見第十二章），讓我們隔離 API 錯誤，並採取適當的動作。在網域偵測到錯誤時，connect-rest 會自動地傳送 500 回應碼，所以接下來你要做的，就是記錄日誌並關閉伺服器。例如：

```
apiOptions.domain.on('error', function(err){
    console.log('API domain error.\n', err.stack);
    setTimeout(function(){
        console.log('Server shutting down after API domain error.');
        process.exit(1);
    }, 5000);
    server.close();
    var worker = require('cluster').worker;
    if(worker) worker.disconnect();
});
```

使用子網域

因為 API 與網站有很大的差異，使用子網域來將 API 與網站其他的部分分隔是很常見的做法。這種做法很容易，我們使用 *api.meadowlarktravel.com*，取代 *meadowlarktravel.com/api* 來重新建構範例。

首先，確保你已安裝 vhost 中介軟體（npm install --save vhost）。在你的開發環境中，或許沒有設定自己的網域名稱伺服器（DNS），所以我們需要欺騙 Express，讓它以為你已連結至一個子網域。為此，我們在**主機檔**裡面添加一個項目。在 Linux 與 OS X 系統中，你的主機檔是 */etc/hosts*，在 Windows 中，它位於 *%SystemRoot%\system32\drivers\etc\hosts* 裡面。如果你的測試伺服器的 IP 位址是 192.168.0.100，可以添加下列幾行到主機檔：

```
192.168.0.100   api.meadowlark
```

如果你是直接在開發伺服器上工作，可以使用 127.0.0.1（*localhost* 的數值），來取代真正的 IP 位址。

現在我們只要連結新的 vhost 來建立子網域：

```
app.use(vhost('api.*', rest.rester(apiOptions));
```

你也需要更改環境：

```
var apiOptions = {
    context: '/',
    domain: require('domain').create(),
};
```

以上就是所有的工作了。在 *api* 子網域，你已經可以使用藉由 rest.VERB 呼叫所定義的所有 API 路由了。