Java 開発 2.0: Groovy の RESTClient を使用して REST によって CouchDB の操作を行う

RESTful な概念とドキュメント指向データベースの動作

Andrew Glover
Author and developer

2009年 11月 17日

この数年におけるオープンソースの世界での急激な技術革新は、Java™ 開発者の生産性を向上させる結果となりました。かつては当たり前に開発に時間がかかっていたツールやフレームワーク、そしてソリューションを、今では無料で入手できるからです。このオープンソースの舞台に前途有望なソリューションとして新たに登場したのが、Apache CouchDB です。Web 2.0 向けデータベースとして一部の人々から支持されている CouchDB は、その全体像を把握しやすく、Web ブラウザーと同じように簡単に使用することができます。連載「Java 開発 2.0」の今回の記事では、CouchDB について紹介するとともに、Groovy の `RESTClient` によって CouchDB の威力を発揮させる方法を説明します。

このシリーズの他の記事を見る

この連載ではこれまで Google のプラットフォームと Amazon のプラットフォームをそれぞれ使用したクラウド・コンピューティングについて説明してきました。この 2 つは実装と構造という点では異なりますが、両方とも素早くスケーラブルなデプロイメントを可能にするという点では同じです。いずれのプラットフォームにしてもこれまでになく素早く、しかも低コストで Java アプリケーションをアセンブルし、テスト、保守することが可能になります。しかし、現在の Java 開発速度に影響を及ぼす要因はクラウドだけではありません。オープンソースのソリューションも、ソフトウェア・アプリケーションの素早いアセンブルに貢献します。なぜなら、これまでのように大量のコードを作成する必要がなくなるからです。自分でオブジェクト・リレーショナル・マッピング (ORM) を作成したり、ロギングやフレームワークのテストを行ったりする時代は、もはや過去の話となりました。これらの問題は、オープンソースの世界ですでに解決されています。しかも、ほとんど必ずと言ってよいほど、オープンソースのソリューションのほうが個人的に作成したソリューションより優れているのが現実です。

この連載について

Java 技術が初めて登場してから現在に至るまでに、Java 開発の様相は劇的に変化しました。成熟したオープンソースのフレームワーク、そしてサービスとして提供される信頼性の高いデプロイメント・インフラストラクチャーを利用できる (借りられる) おかげで、今では Java アプリケーションを短時間かつ低コストでアセンブルし、テスト、実行、保守することが可

能になっています。この連載では Andrew Glover が、この新たな Java 開発パラダイムを可能にする多種多様な技術とツールを詳しく探ります。

Java 開発全般にわたり、オープンソースの技術革新がアプリケーションのアセンブルを容易にしています。オープンソースのデータベースとして新たに登場した Apache CouchDB (この記事を執筆している時点では、リリース 0.10.0) も、その例外ではありません。CouchDB をいったん稼働させれば、後の操作はごく簡単です。しかも CouchDB を操作する上で必要なのは HTTP 接続だけです。JDBC ドライバーも、サードパーティー製の管理プラットフォームも一切必要ありません。この記事では CouchDB について紹介し、このデータベースを短時間で使いこなせるようになる方法を説明します。その方法とは、CouchDB を簡単にインストールできるように Amazon の EC2 プラットフォームを利用し、使い勝手の良い Groovy モジュールによって CouchDB と通信するというものです。

ドキュメント指向のデータベース

データベース市場を支配しているのは基本的にリレーショナル・データベースですが、時には他のタイプのデータベースを使ったほうが理にかなうこともあります。それは例えば、オブジェクト指向のデータベースやドキュメント指向のデータベースなど、リレーショナル指向の世界とはかけ離れたデータベースであったりします。CouchDB は、ドキュメント指向のデータベースに属します。このデータベースにはスキーマがなく、文書を JSON (JavaScript Object Notation) ストリングという形で保管することができます。

JSON

JSON は軽量の、Web アプリケーションにも使用できるデータ交換フォーマットです。JSON は XML と似ていますが、XML よりも遥かに簡潔です。その軽量さのおかげで、JSON は Web の共通語になりつつあります。JSON についての詳細は、「参考文献」を参照してください。

例えば、駐車違反切符について考えてみてください。この紙切れには、以下を含め、さまざまな項目が記載されます。

- 違反した日付
- 時刻
- 場所
- 自動車についての説明
- ナンバー・プレートの情報
- 違反内容

違反切符のフォーマットとそこに記入されるデータは、管轄ごとに異なることがあります。1 つの管轄内での標準的な駐車違反切符でさえも、何が記入され、何が記入されていないかは個々の切符によってたいがい異なるはずです。例えば、違反切符を切る警官が時刻を記入しなかったり、自動車のメーカーとモデルの代わりにナンバー・プレートの情報だけを記入したりすることもあるでしょう。また、場所についての情報が 2 本の通り (Street) の組み合わせとして、「intersection of Fourth and Lexington (4 番街と Lexington 通りの交差点)」のように記入されることも、「19993 Main Street (19993 Main 通り)」のように確定的な住所で記入されることも考えられます。しかし、記入されている大まかな内容はほとんど変わりありません。

違反切符のデータのポイントは、リレーショナル・データベースでモデル化することができますが、その詳細となると少し難しくなります。例えば、リレーショナル・データベースに交差点の場所を格納する方法を考えてみてください。(個々の列に明確な通りの名前を格納するようにモデル化した場合)、交差する通りがなかったとしたら、データベースの 2 番目の住所のフィールドは空白になるのでしょうか。

このようなケースは、リレーショナル・データベースで扱うには少し難解すぎるかもしれません。必要な情報がすでに文書 (違反切符) という形で存在しているのであれば、なぜデータを文書としてモデル化しないのでしょうか。データを文書としてモデル化すれば、厳密なリレーショナル・モデルに必ずしも合わせることなく、概略モデルのセマンティクスに大まかに従うことができます。そこで登場するのが、CouchDB です。CouchDB ではこのようなケースを、自己完結した文書として柔軟にモデル化することができます。この文書にスキーマは含まれませんが、代わりに他の文書にもほとんど共通するブループリントが含まれます。

MapReduce

Google によって開発された MapReduce は、巨大なデータ・セットの処理を目的とした概念フレームワークです (「参考文献」を参照)。このフレームワークは多数のコンピューターを用いた分散処理で問題を解決するように高度に最適化されています。MapReduce は、Map と Reduce という 2 つの関数の組み合わせで、まず Map 関数によって大量の入力データを受け取り、それを分割します (そして、そのデータを他のプロセスに渡して処理させます)。続いて Reduce 関数で、Map から個別に返されたすべての結果を 1 つの最終的な出力に集約するという仕組みです。

CouchDB では、文書そのものや、文書のプロパティーを検索できるだけでなく、リレーショナルでの世界と同じように文書を関連付けることもできます。けれどもそのための手段は SQL ではなく、ビュー (views) を使用します。ビューは基本的に、MapReduce の様式で (JavaScript を使用して) 作成する関数です。つまり、最終的には Map 関数と Reduce 関数を作成することになります。この 2 つの関数が連動して、極めて効率的に文書からデータを抽出したり、文書間の関係を活用したりするわけです。実際、ベースとなる文書が変更されていなければ、CouchDB はこれらの関数を 1 度実行すればよいだけの賢さを持ち合わせているため、ビューは極めて高速な動作をします。

CouchDB でとりわけ興味深い点は、その設計です。CouchDB は Web そのものが持つ基本的な (そして極めて成功した) 概念を具現化しており、文書、ビュー、データベースを対象に、作成、照会、更新、そして削除を行える、完全に RESTful な API を公開します。そのため、CouchDB は極めて簡単に使いこなせるようになります。開発に取り掛かるのに、ドライバーや他のプラットフォームは必要ありません。ブラウザーさえあれば十分です。とは言え、CouchDB の操作をさらに容易にするライブラリーも多数揃っています。ただし、これらのライブラリーは HTTP を介して RESTful な概念を利用しているだけにすぎません。

Web そのものにかなりよく似た CouchDB は、スケーラブルに作成されています。その作成に使用されている言語は、耐障害性を備えた無停止分散アプリケーションの構築をサポートする並行処理プログラミング言語、Erlang です。現在オープンソースとして使用できるこの言語は、Ericsson によって開発され、通信の分野で広く利用されてきました。

CouchDB をクラウド・スタイルでインストールする

CouchDB のインストールは、オペレーティング・システムによってさまざまに異なります。Windows® を使用している場合には、Cygwin と Microsoft C コンパイラー、そして関連するその他多数の依存関係が必要になります。Mac を使用している場合には、Macports が必要です。一方 Linux® プラットフォーム、例えば Ubuntu を実行しているとしたら、インストールはこれ以上なく簡単になります。しかし、誰もが Ubuntu インスタンスを使用しているわけではありません。皆さんは Ubuntu を使用していますか？

もちろん、Ubuntu インスタンスを使用するのは簡単です。というのも Amazon の EC2 では、オンデマンドで使用できる Ubuntu を比較的低価格で提供しているからです。つまり、言ってみればちょっとした EC2 のマジックによって CouchDB を即座に稼働状態にし、作業が終わったら、パワー・ダウンすればよいわけです。

まずは、基本インスタンスとして機能する EC2 AMI を見つける必要があります。私は結局、Ubuntu 9.04 のインスタンスである AMI ami-ccf615a5 を使用することにしました。これは、この記事を執筆している時点での最新バージョンです (皆さんがこの記事を読む頃には、これよりも新しいバージョン 9.10 の AMI が使用可能になっているはずです)。Eclipse または AWS Management Console を使用して、ami-ccf615a5 のインスタンスを起動します。セキュリティー・ポリシーは必ず、SSH によるアクセスを許可するように設定してください (CouchDB は HTTP を使用しますが、ここでは単純化を図って通信には SSH トンネルを使用します)。また、キー・ペアを使用する必要もあります (そのための説明が必要な場合は、この連載の 2 つの記事、「EC2 も借りられます」と「容易な EC2」を参照してください)。

Ubuntu 9.04 の EC2 インスタンスを起動したら、`ssh` でインスタンスに接続します (インスタンスが完全に起動するまでには 1 分程度かかる場合もあることに留意して、辛抱強く待ってください)。例えば私の環境では、以下のコマンドで端末から新しく作成されたインスタンスに `ssh` で接続することができます。

```
aglover#> ssh -i .ec2/agkey.pem root@ec2-174-129-157-167.compute-1.amazonaws.com
```

私が持っている AMI の DNS 名は ec2-174-129-157-167.compute-1.amazonaws.com で、参照しているのは agkey という名前のキー・ペアです。皆さんの DNS 名とキー・ペアは、もちろんこれとは異なります。

クラウド化された Ubuntu インスタンスでコマンド・プロンプトが出されたら、以下のように入力します。

```
apt-get update
```

続いて以下のように入力します。

```
aptitude install couchdb
```

上記の 2 つのコマンドによって、CouchDB が自動的にインストールされます。ただし、最新のバージョンがインストールされるわけではないことに注意してください。確実に最新のバージョン

をインストールしなければならない場合は、CouchDB をソースからインストールする必要があります (「参考文献」を参照)。

上記のコマンドの実行が完了した後、`ps -eaf` コマンドを実行して CouchDB が稼働しているかどうかを確かめることができます。`couchdb` によって実行しているいくつかのプロセスのパスを調べるには、`ps` 出力を `egrep` にパイプします。すると、リスト 1 に記載するような出力行が表示されます。

リスト 1. 稼働中の CouchDB (記事のページ幅に合わせて改行してあります)

```
couchdb   1820     1  0 00:54 ?        00:00:00 /bin/sh -e /usr/bin/couchdb
   -c /etc/couchdb/couch.ini -b -r 5 -p /var/run/couchdb.pid -o /
couchdb   1827  1820  0 00:54 ?        00:00:00 /bin/sh -e /usr/bin/couchdb
   -c /etc/couchdb/couch.ini -b -r 5 -p /var/run/couchdb.pid -o /
couchdb   1828  1827  0 00:54 ?        00:00:00 /usr/lib/erlang/erts-5.6.5/bin/beam
   -Bd -- -root /usr/lib/erlang -progname erl -- -home /v
couchdb   1836  1828  0 00:54 ?        00:00:00 heart -pid 1828 -ht 11
```

次に、ローカル・マシンに戻って SSH トンネルをセットアップします。この SSH トンネルによって、クラウドで稼働中の CouchDB インスタンスに、あたかも自分のマシン上に CouchDB インスタンスが常駐しているかのようにアクセスできるようになります。それにはまず、ローカル・マシンで新しいターミナル・セッションを開始し、以下の内容を入力します。

```
ssh -i your key -L 5498:localhost:5984 root@your AMI DNS
```

ローカル・マシン上でブラウザーを起動し、アドレス・バーに `http://127.0.0.1:5498/` と入力します。以下のような JSON フォーマットのウェルカム・メッセージが表示されるはずです。

```
{"couchdb":"Welcome","version":"0.8.0-incubating"}
```

すべてが順調に機能するようになったところで、次は CouchDB の機能を試す番です。

Groovy の `RESTClient` で RESTfull に操作する

REST

REST (Representational State Transfer) とは、疎結合された Web アプリケーションの設計スタイルです。これらの Web アプリケーションはメッセージに依存するのではなく、URL (Uniform Resource Locator)、URI (Uniform Resource Identifier)、URN (Uniform Resource Name) などの形で名前が付けられたリソースに依存します。REST は、すでにその実力が実証済みで Web のインフラストラクチャーとして成功している HTTP を巧みに利用します。つまり、REST は GET や POST リクエストなどの HTTP プロトコルの側面を利用するということです。これらのリクエストは、CRUD (Create、Read、Update、Delete) などの標準的なビジネス・アプリケーションのニーズに見事に対応します。

CouchDB は RESTful な HTTP インターフェースを介してデータを公開するため、(すでにブラウザーを使った操作でわかったように) CouchDB を操作するのは至って簡単です。必要な操作のほとんどは、HTTP を介して行うことができます。

HTTP を扱うための対話型のツールには数々の選択肢があります。RESTful なインターフェースを操作するときに私が好んで使っているツールの 1 つは、Groovy の HTTPBuilder を拡張した `RESTClient` です (「参考文献」を参照)。よく使われている Apache Commons Project の HTTPClient

をラップする HTTPBuilder は、HTTP の POST、GET、PUT、DELETE の構文に、巧妙な Groovy の特性を付加します。HTTPBuilder は Groovy でビルドされ、Groovy を利用することから、RESTful な概念 (CouchDB との通信もその一例です) を利用するスクリプトをこれ以上ないほど簡単に作成することができます。

Grape で容易かつ迅速に行う

速い、簡単、しかも無料 (または安価) という Java 開発 2.0 全体のテーマに従うとすれば、HTTPBuilder のようなライブラリーを扱うときにとりわけ関連してくるのが、Groovy の便利な機能である Grape (Groovy Advanced Packaging Engine または Groovy Adaptable Packaging Engine) です (「参考文献」を参照)。依存性マネージャーである Grape によって、Groovy のスクリプトとクラスは、実行時にその特定の依存関係を自動で構成できるようになります。つまりコードの作成に取り掛かるために一連の JAR ファイルをダウンロードする必要がなくなるため、各種のオープンソース・ライブラリーをごく簡単に使用できるということです。例えば Grape を使用すれば、HTTPBuilder を使用する Groovy スクリプトを作成する前に、HTTPBuilder に必要な JAR を用意する必要はありません。これらの必要な JAR は、Grape によって実行時 (またはコンパイル時) に、(Apache Ivy により) ダウンロードされます。

Grape を使用するために用いる手段は、アノテーションとメソッド呼び出しです。例えば `@Grab` アノテーションでメソッドまたはクラス宣言を修飾し、このアノテーションのなかに、主要な依存関係に関するメタデータを指定します (Ivy のマジックにより、過渡的なすべての依存関係も判断されます)。実行時またはコンパイル時 (どちらか先に行われるほう) に、Grape はこれらの依存関係をダウンロードし、確実にクラス・パスに組み込みます。(例えば以前に実行したときなどに) すでに依存関係がダウンロードしてある場合でも、Grape は適切な JAR ファイルが確実にクラス・パスに組み込まれるようにします。

Groovy を使用して REST によって CouchDB に容易に文書を作成する

CouchDB 内に文書を作成するには、まずその前にデータベースを作成しなければなりません。例えば駐車違反切符のデータベースを作成する場合、リスト 2 に記載するように HTTPBuilder の巧妙なドメイン特化言語 (DSL) で、`RESTClient` を使って HTTP PUT を実行します (この記事に記載するすべての Groovy サンプル・コードは、ダウンロードすることができます)。

リスト 2. CouchDB データベースの作成

```
import static groovyx.net.http.ContentType.JSON
import groovyx.net.http.RESTClient

@Grab(group='org.codehaus.groovy.modules.http-builder', module='http-builder',
   version='0.5.0-RC2')
def getRESTClient(){
  return new RESTClient("http://localhost:5498/")
}

def client = getRESTClient()
def response = client.put(path: "parking_tickets",
        requestContentType: JSON, contentType: JSON)

assert response.data.ok == true : "response from server wasn't ok"
```

CouchCB は `{"ok":true}` というレスポンスを返すはずです。リスト 2 を見るとわかるように、HTTPBuilder では簡単に JSON を構文解析し、`ok` 要素の値が本当に `true` であることを確認することができます。

次は、駐車違反切符のテーマに沿っていくつかの文書を作成します。駐車違反切符をモデル化する際には、切符にはさまざまな側面が関連付けられることを思い出してください。また、これらの文書は警官が記入する実際の書式であるため、一部のフィールドが記入されていないことや、さらには所定のパターンに従っていないこともあるという点にも注意してください。例えば前に説明したように、場所は交差点として記入される場合もあれば、実際の住所が記入される場合もあります。

HTTPBuilder を使用する場合、HTTP PUT によって CouchDB 内に文書を作成することができます (リスト 2 でデータベースを作成したときと同じ方法です)。CouchDB では JSON 文書を扱うので、JSON の名前と値のフォーマットに従わなければなりません。そのためには、Groovy でマップのようなデータ構造を作成します (このデータ構造を、HTTPBuilder が有効な JSON に変換します)。リスト 3 にその方法を記載します。

リスト 3. `RESTClient` による CouchDB 文書の作成

```
response = client.put(path: "parking_tickets/1234334325", contentType: JSON,
        requestContentType:  JSON,
        body: [officer: "Kristen Ree",
                location: "199 Baldwin Dr",
                vehicle_plate: "Maryland 77777",
                offense: "Parked in no parking zone",
                date: "2009/01/31"])

assert response.data.ok == true : "response from server wasn't ok"
assert response.data.id == "1234334325" : "the returned ID didn't match"
```

リスト 3 で何が行われているかを説明すると、まず、CouchDB 文書のための PUT を実行するときに UUID を割り当てています。UUID は、CouchDB に自動的に割り当てさせることも、自分で管理することもできます。リスト 3 では `1234334325` という仮の UUID を作成しました。この UUID が結果的に URL に追加されることになります。該当する UUID がある場合、CouchDB はその UUID に PUT で作成した文書を割り当てます。上記の `put` 呼び出しの `body` では、通常のマップとほとんど同じように、それぞれの名前に値が関連付けられている点に注目してください。例を挙げると、担当警察官の名前は `Kristen Ree`、違反切符の場所は `199 Baldwin Dr` となっています。

リスト 4 では、同じ手法を使って CouchDB に別の駐車違反切符を作成しています。

リスト 4. 別の駐車違反切符

```
def id = new Date().time
response = client.put(path: "parking_tickets/${id}", contentType: JSON,
        requestContentType:  JSON,
        body: [officer: "Anthony Richards",
                location: "Walmart Parking lot",
                vehicle_plate: "Delaware 4433-OP",
                offense: "Parked in non-parking space",
                date: "2009/02/01"])


assert response.data.ok == true : "response from server wasn't ok"
assert response.data.id == "${id}" : "the returned ID didn't match"
```

上記では、`RESTClient` によって PUT を実行するたびに、JSON レスポンスに `ok` の値として true が含まれることをアサートし、対象の `id` 値が存在することを確認しています。このリスト 4 では、UUID を作成する代わりに、現在時刻を使用していることに注意してください。これは絶対確実な方法とは言えませんが、単純な対話にはこれで十分です。

CouchDB では新規文書の作成が正常に完了すると、UUID とリビジョン ID が含まれる JSON をレスポンスとして返します。例えば以下のレスポンスは、リスト 4 で検証対象としている JSON を表しています。

```
{"ok":true,"id":"12339892938945","rev":"12351463"}
```

皆さんの `id` と `rev` の値は上記とは確実に異なります。`id` 値は、`response.data.id` のような呼び出しを実行することで取得できることに注意してください。

CouchDB 内では、文書はリビジョンによって追跡されます。そのため、CVS やSubversion の場合と同じように、リビジョン ID を使って前の文書バージョンに戻すことができます。

CouchDB でのビュー

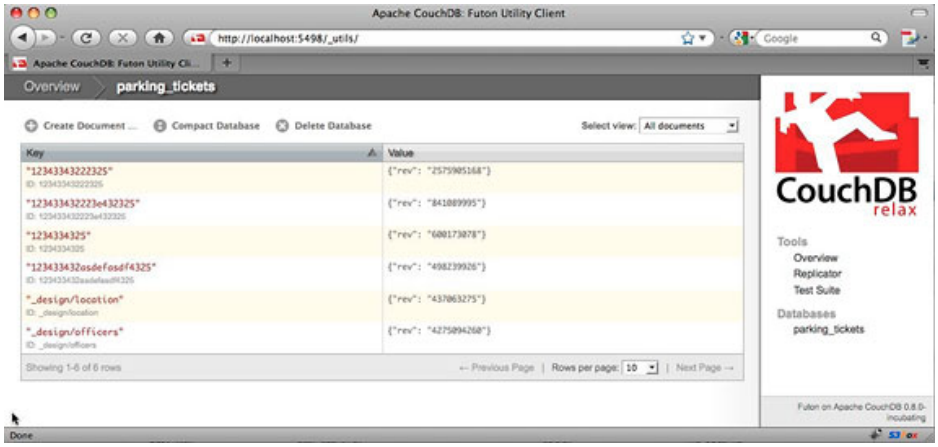
駐車違反切符 (つまり、CouchDB で言う文書) をいくつか作成したので、今度は CouchDB でビューを作成します。ビューとは単に MapReduce 関数を実行した結果であることを思い出してください。したがって、これらの関数を定義する必要があります。しかし多くの場合 Reduce 関数を定義する必要はありません。もう一方の Map 関数がほとんどの作業を自動的に処理してくれるからです。Map 関数の実行内容はその名のとおりで、基本的に、この関数によってフィルタリングや検索の対象とする「もの」または側面をマップ (分割して分散処理) することができます。

これまでの作業で 2 つの違反切符を定義しました。1 つは警官 Ree が発行した違反切符、もう 1 つは警官 Richards が発行した違反切符です。例えば警官 Ree が発行したすべての違反切符を検索するには、警官に応じて `officer` プロパティーをフィルタリングする Map 関数を作成します。そしてこの関数の結果を CouchDB の `emit` 関数に渡します。

CouchDB の管理インターフェースである Futon を使用する

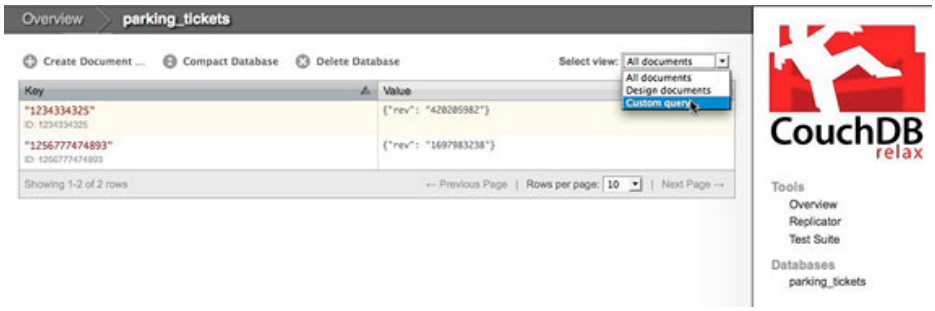
ビューを定義するには、CouchDB の RESTful な API を使うことも、あるいは Futon と呼ばれる CouchDB の管理インターフェースを使うこともできます。Futon は、http://localhost:5498/_utils/ にアクセスすると使用できる Web アプリケーションです。このインターフェースに早速アクセスしてみてください。これまでの手順に従ってデータベースと文書を作成してあるとすれば、parking_tickets データベースを対象とした単純なインターフェースが表示されます (図 1 を参照)。

図 1. Futon インターフェース



parking_tickets データベースを選択すると、ページの右端にドロップダウン・リスト (「Select view」という名前のリスト) が表示されます。カスタム・ビューを定義するには、このリストから「Custom query...」を選択します (図 2 を参照)。

図 2. Futon のビュー選択インターフェース



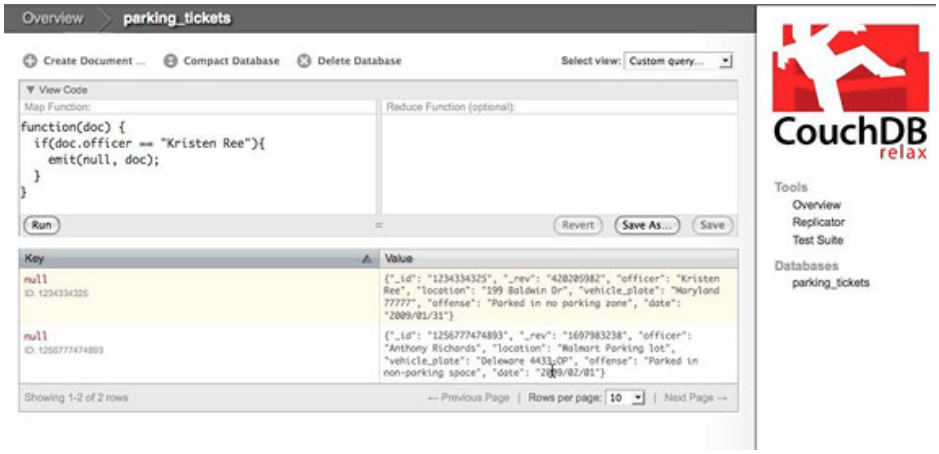
これで現在、Futon には Map 関数と Reduce 関数の両方を定義するためのインターフェースが表示されているはずです (「View code」リンクをクリックする必要がある場合もあります)。「Map」テキスト・ボックスに、リスト 5 に記載する単純な Map 関数を定義してください。

リスト 5. CouchDB の単純な Map 関数

```
function(doc) {
  if(doc.officer == "Kristen Ree"){
    emit(null, doc);
  }
}
```

ご覧のように、リスト 5 の Map 関数は JavaScript で定義されています。この関数は、CouchDB データベース内の文書を、文書の `officer` プロパティーを基準にフィルタリングするだけにすぎません。具体的には、文書に設定されている警官の名前が `Kristen Ree` である場合にのみ、その文書を `emit` 関数に渡します。図 3 に、Futon のどこにこの関数を定義したかを示します。

図 3. MapReduce 関数の作成



続いて、設計文書の名前とビューの名前を入力するよう求められます (それぞれ `by_name`、`officer_ree` と入力してください)。これらの名前が、後でこのビューを呼び出すための URL を構成します (したがって、URL は http://localhost:5498/parking_tickets/_view/by_name/officer_ree となります)。

これで、このビューを HTTPBuilder で使用できるようになりました (リスト 6 を参照)。

リスト 6. 新規ビューの呼び出し

```
response = client.get(path: "parking_tickets/_view/by_name/officer_ree",
        contentType: JSON,  requestContentType:  JSON)

assert response.data.total_rows == 1

response.data.rows.each{
   assert it.value.officer == "Kristen Ree"
}
```

このビューは、1 つの文書だけが含まれる JSON レスポンスを適切に返します。その文書とはすなわち、1月31日に警官 Ree が発行した違反切符です。リスト 6 の `response` オブジェクトは JSON を適宜構文解析するので、解析処理をする前の HTTP レスポンスは表示されません。処理前の JSON レスポンスを表示するには、`response` オブジェクトの `data` プロパティーで `toString` メソッドを呼び出します。処理前のレスポンスはリスト 7 のように表示されます。

リスト 7. ビューの処理前の結果

```
{"total_rows":1,"offset":0,"rows":[
  {"id":"1234334325","key":null,
   "value":{"_id":"1234334325","_rev":"4205717256","officer":"Kristen Ree",
       "location":"199 Baldwin Dr","vehicle_plate":"Maryland 77777",
       "offense":"Parked in no parking zone","date":"2009/01/31"}}]}
```

返された JSON 文書が処理される前の状態を見るとわかるように、JSON を難なく構文解析する HTTPBuilder の機能は、各種の属性とそれに対応する値を評価するためのオブジェクト・グラフ的なメカニズムを実現することから、極めて重宝します。

説明のため、これからデータベースにさらに文書を追加します。以降の例で説明する手順に従うには、コードのダウンロードを使って同じように文書を追加してください。

CouchDB の `emit` 関数は、オーガナイザーのような役割を果たします。Map 関数に制約を設けない場合 (リスト 5 では制約を設けました)、渡された文書を `emit` 関数がソートすることになります。例えば、特定の日付の違反切符をすべて取得するには (SQL の`ORDER BY` 節を考えてください)、文書の `date` フィールドを指定して `emit` を実行すればよいだけです (リスト 8 を参照)

リスト 8. さらに単純になった Map 関数

```
function(doc) {
  emit(doc.date, doc);
}
```

リスト 9 は、このビューに対して HTTP GET を実行します (設計文書の名前には `dates` を、ビューの名前には `by_date` を指定してあります)。

リスト 9. 別のビューの呼び出し

```
response = client.get(path: "parking_tickets/_view/dates/by_date", contentType: JSON,
        requestContentType:  JSON)
assert response.data.total_rows == 4
```

リスト 9 のクエリーは、parking_tickets データベースに保管されたすべての文書を、日付順にソートして返します。`assert` 文は、単に `total_rows` プロパティーが `4` であることを確認しているだけにすぎません。重要なポイントは、ビューは結果と併せて多少のメタデータ (返された文書の数など) を返すことです。このメタデータが、構文解析を始める前に、処理前のレスポンスを確認するのに役立ちます。リスト 10 に、処理前の結果を記載します。

リスト 10. 日付順にソートされた処理前の JSON 文書

```
{"total_rows":4,"offset":0,"rows":[
  {"id":"85d4dbf45747e45406e5695b4b5796fe","key":"2009/01/30",
   "value":
    {"_id":"85d4dbf45747e45406e5695b4b5796fe","_rev":"1318766781",
     "officer":"Anthony Richards",
     "location":"54th and Main","vehicle_plate":"Virginia FCD-4444",
     "offense":"Parked in no parking zone","date":"2009/01/30"}},
  {"id":"1234334325","key":"2009/01/31",
   "value":
    {"_id":"1234334325","_rev":"4205717256",
     "officer":"Kristen Ree",
     "location":"199 Baldwin Dr","vehicle_plate":"Maryland 77777",
     "offense":"Parked in no parking zone",
     "date":"2009/01/31"}},
  {"id":"12345","key":"2009/01/31",
   "value":
    {"_id":"12345","_rev":"1479261876",
     "officer":"Anthony Richards","location":"1893 Main St",
     "vehicle_plate":"Maryland 4433-OP",
     "offense":"Parked in no parking zone","date":"2009/01/31"}},
  {"id":"12339892938945","key":"2009/02/01",
   "value":
    {"_id":"12339892938945","_rev":"12351463","officer":"Anthony Richards",
     "location":"Walmart Parking lot","vehicle_plate":"Maine 4433-OP",
     "offense":"Parked in non-parking space",
    "date":"2009/02/01"}}]}
```

ここで興味深いのは、このようにビューを定義することで、キーを渡せるようになるという点です。つまり、`emit` 関数の最初の値で必ず表現したいものを渡すことができます。例えばリスト 8 に定義したビューは、基本的に日付順にソートを行いますが、特定の日付でソートする場合に

は、その日付をビューのクエリーに渡せばよいというわけです。試しに、以下の URL をブラウザーのアドレス・バーに入力してみてください。

```
http://localhost:5498/parking_tickets/_view/dates/by_date?key="2009/01/31"
```

このビューによって返されるのは 1月31日に発行された違反切符だけです。ブラウザーのウィンドウには、リスト 11 に記載するような JSON 風のテキストが表示されます。このように、ブラウザーをクエリー・ツールとして使用すると、HTTP リクエストに対する JSON レスポンスの処理前の内容を特に簡単に確認できることがわかります。

リスト 11. 1月31日に発行された 2 つの違反切符

```
{"total_rows":4,"offset":1,"rows":[
   {"id":"1234334325","key":"2009/01/31",
    "value":
     {"_id":"1234334325","_rev":"4205717256","officer":"Kristen Ree",
       "location":"199 Baldwin Dr","vehicle_plate":"Maryland 77777",
       "offense":"Parked in no parking zone",
       "date":"2009/01/31"}},
   {"id":"12345","key":"2009/01/31",
    "value":
     {"_id":"12345","_rev":"1479261876","officer":"Anthony Richards",
       "location":"1893 Main St","vehicle_plate":"Maryland 4433-OP",
       "offense":"Parked in handicap zone without permit",
        "date":"2009/01/31"}}]}
```

ビューは、いくらでも具体的にすることができます。例えばリスト 12 のように、ちょっとした JavaScript ストリング操作を使用することによって、Main Street のいずれかの場所で発行された違反切符を検索するビューを作成することができます。

リスト 12. ストリングのマジックを使用した別のビュー

```
function(doc) {
  if(doc.location.toLowerCase().indexOf('main') > 0){
   emit(doc.location, doc);
  }
}
```

リスト 12 からわかるように、`location` 要素に `main` が含まれている文書はすべて、`emit` 関数に渡されます。ここで念頭に置いておかなければならない点は、この検索はかなり広範なものであるということです。文書の「場所 (`location`)」欄に `Germaine Street` といったストリングが含まれている場合も、その文書が返されることになります。私が定義した違反切符には記入箇所がわずかしかないので、このビューが返す結果はリスト 13 のようになります。

リスト 13. location に main が含まれていることを基準にフィルタリングした結果

```
{"total_rows":2,"offset":0,"rows":[
  {"id":"123433432asdefasdf4325","key":"4th and Main",
   "value":
    {"_id":"123433432asdefasdf4325","_rev":"498239926",
    "officer":"Chris Smith","location":"4th and Main",
      "vehicle_plate":"VA FGA-JD33",
      "offense":"Parked in no parking zone","date":"2009/02/01"}},
 {"id":"123433432223e432325","key":"54 and Main",
   "value":
    {"_id":"123433432223e432325","_rev":"841089995",
    "officer":"Kristen Ree","location":"54 and Main Street",
      "vehicle_plate":"Maryland 77777",
      "offense":"Parked in no parking zone","date":"2009/02/02"}}]}
```

上記の JSON レスポンスには `key` 要素が含まれていることに注目してください。これは、その特定の文書が検出された理由を記述する要素で、このレベルの情報はかなり役に立ちます。また、私が定義したさまざまな違反切符のデータには、多少一貫性が欠けていることも注目の点です。例えば、場所 (location) が正確に記述されているものも、そうでないものもあります。このようなデータはリレーショナル・データベースには保管できないものの、ドキュメント指向のモデルであれば上手く適合すると思いませんか？しかも、Groovy の威力、そして JSON をいとも簡単に構文解析する HTTPBuilder の機能によって、極めて簡単にデータを見つけ出すことができます (そのままの JDBC を使うより遥かに簡単です)。

Web 用データベースとしての CouchDB

CouchDB で特に興味を引かれるのは、その操作の容易さです。リレーショナル・データベースも同じく簡単に使用できますが、CouchDB の良いところは、例えば Web ブラウザーを使い慣れているとしたら、Web ブラウザーを使ってその API を利用できることです。その上 CouchDB の RESTful な API により、HTTPBuilder の `RESTClient` のような賢いフレームワークを介して通信することもできます。さらに、使用できるのは HTTPBuilder だけではありません。他にも多数の Java ライブラリーが CouchDB の操作を簡易化しようと試みています。そのうち特に前途有望なのは、jcouchdb です (「参考文献」を参照)。jcouchdb では、CouchDB が持つ RESTful な部分と JSON の部分に触れることなく、Java 言語でのプログラムによって文書およびビューを操作できるようにします。

次回の記事では、再び Google App Engine を取り上げます。オープンな技術革新の精神にのっとり、Google App Engine の開発およびデプロイメントを容易にする新しいフレークワークが次々と登場しています。そのなかから、Google のクラウド・プラットフォームでの Java 開発 2.0 をさらに容易にする 1 つのフレームワークを取り上げます。

ダウンロード

内容	ファイル名	サイズ
Groovy source code for article examples	j-javadev2-5.zip	2KB

著者について

Andrew Glover

Andrew Glover は、ビヘイビア駆動開発、継続的インテグレーション、アジャイル・ソフトウェア開発に情熱を持つ開発者であるとともに、著者、講演者、起業家でもあります。詳細は彼のブログにアクセスしてください。