KAPITOLA 3

Uchování dat

V této kapitole:
- Proč MongoDB?
- Shrnutí

„Pozor na chyby v tom kódu; já jsem pouze dokázal, že funguje, nezkoušel jsem ho."

Donald E. Knuth

Proč MongoDB?

V této kapitole se naučíte používat MongoDB. MongoDB je typ NoSQL – dokumentově orientovaného databázového systému. Ekosystém Node.js podporuje spoustu oblíbených relačních databází, jimiž jsou kupříkladu PostgreSQL nebo MySQL. Relační databáze jsou robustní a časem prověřené a dobře by vyhovovaly požadavkům moderních webových aplikací. Proč byste tedy měli používat databázový systém MongoDB? Abyste získali odpověď na tuto otázku, přečtěte si o jeho nejdůležitějších vlastnostech.

Hlavní vlastností MongoDB je, že se jedná o dokumentově orientovaný systém bez schématu. Dokumentem, jenž je obsažen v termínu **dokumentově orientovaný**, je BSON dokument. Jedná se o zkratku pro **binární JSON** dokument (Binary JSON).[1] Označení **bez schématu** znamená, že neexistuje předem definovaná struktura. MongoDB si můžete jednoduše představit jako obří pole objektů s rychlými funkcemi pro vkládání a vyhledávání.

Pro zastánce relačních databází může být databáze bez schématu něco jako vetřelec. Existují ale situace, v nichž MongoDB nabízí řadu výhod oproti relačním databázím.

V prvé řadě se skvěle hodí pro postupný vývoj. Představte si, že máte tabulku `Person` (osoba) a chcete do ní přidat seznam oblíbených filmů. Ve světě relačních databází byste museli přidat tabulku `Movie` a propojit obě tabulky, abyste mohli ukládat vztahy. Ve světě MongoDB můžete jednoduše přidat pole filmů k tabulce `Person` v závislosti na požadavcích. U startupů s proměnlivými požadavky a rozrůstajícím se týmem je agilita MongoDB důležitou vlastností.

MongoDB se také skvěle hodí pro nestrukturovaná data. Ve světě financí představují deriváty složité téma – jsou jich tisíce typů (například forwardy, opce a swapy), přičemž každý

1 *http://bsonspec.org/*

z nich má specifické vlastnosti. Promítnout je do relační databáze by bylo velmi náročné, ale v databázi MongoDB stačí převést derivát do formátu JSON a vložit do databáze. Deriváty můžou být libovolně složité a můžete je uložit vedle sebe do kolekce.

Databázový systém MongoDB byl od začátku navržen pro ukládání velkých objemů dat. K tomu podporuje mechanizmus označovaný jako **lámání** (**sharding**). Obvyklá struktura knih na serveru Yellow Pages[2] je jednoduchým příkladem – knihy bývají rozdělené na dva úlomky, a to A–K a L–Z. Vyhledávání informací je rychlejší, protože si vyberete příslušný úlomek – menší objem dat. Koncepce lámání v databázích je stejná, ačkoliv implementace je složitější.

Další případy užití MongoDB najdete na oficiálních stránkách.[3] MongoDB se navíc stává standardním databázovým systémem pro prostředí Node.js. Jason Hoffman, zakladatel společnosti Joyent, prezentoval Node.js a MongoDB jako moderní kombinaci pro real-time web,[4] která by měla nahradit tradiční kombinaci LAMP (Linux, Apache, MySQL a PHP). Až začnete vytvářet skutečné aplikace, nevyhnutelně narazíte na MongoDB a budete moct použít dovednosti získané z této knihy.

Instalace

Nyní se dozvíte, jak přejít z hostovaného MongoDB na lokálně nainstalovanou verzi. Instalace je poměrně jednoduchá, přičemž poslední verzi tohoto systému si můžete stáhnout na internetové adrese *https://www.mongodb.org/downloads*. Až najdete odkaz pro svou platformu, stáhněte a rozbalte si příslušných archiv.

Níže uvedený návod funguje pro systémy Debian a Ubuntu. Postup pro ostatní platformy najdete v oficiální dokumentaci.[5] Měli byste zadat následující příkazy:

```
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.2.4.tgz
tar –zxvf mongodb-linux-x86_64-3.2.4.tgz
```

Potom zbývá už jen vytvořit adresář *data/db*:

```
sudo mkdir /data
sudo mkdir /data/db
```

Nakonec spustíte démona MongoDB:

```
sudo mongodb-linux-x86_64-3.2.4/bin/mongod
```

Pokud proběhlo vše v pořádku, uvidíte následující (nebo podobný) výpis v konzoli:

```
...
[initandlisten] waiting for connections on port 27017
[websvr] admin web console waiting for connections on port 28017
```

2 *http://www.yellowpages.com/*

3 *https://docs.mongodb.org/ecosystem/use-cases*

4 *https://www.mongodb.com/presentations/n2m-nodejs-and-mongodb-modern-stack-real-time-web*

5 *https://docs.mongodb.org/manual/installation/*

Výběr ovladače

Pro prostředí Node.js existuje spousta modulů MongoDB. Dvě nejoblíbenější řešení představují modul Mongoose a nativní ovladač.

Mongoose mapuje objekty na dokumenty – stručně jsme si ho předvedli v kapitole 1. Umožňuje definovat schémata v prostředí Node.js, přičemž ta převádí na dokumenty BSON pro MongoDB, a přitom abstrahuje některé nízkoúrovňové implementační detaily. To je výhodné, jelikož abstrakce urychluje vývoj aplikací. Nevýhodou je snížení výkonu v určitých situacích – kvůli veškeré magii, která se odehrává na pozadí.

Alternativou je nativní ovladač, což je rovněž ovladač oficiálně podporovaný společností 10gen, která vyvíjí MongoDB. S nativním ovladačem musíte pracovat s nízkoúrovňovými operacemi – například zakládat asynchronní spojení s databází. Musíte také definovat, jak nakládat se schématy na aplikační úrovni.

Operace databáze MongoDB se velmi podobají operacím nativního ovladače, takže vývojáři se nemusí učit příliš mnoho různých funkcí. Kromě toho poskytuje nejlepší výkon. Oba ovladače jsou perfektní, ale ve zbytku této knihy se budete setkávat s nativním ovladačem. Jakmile pochopíte základy nativního ovladače, můžete poměrně snadno přejít na Mongoose, jestliže lépe vyhovuje vašim požadavkům.

Rychlý test

Pro tuto kapitolu si dáme první cíl – napsat modul, jenž umí:

- připojit se k databázi MongoDB,
- vložit obchodní transakci,
- načíst transakci.

Začněme testy jednotek – budeme stavět na tom, co jsme se naučili v předchozí kapitole. Uložíme níže uvedený kód do souboru *test/db.test.js*:

kapitola03/test/db.test.js (*úryvek*)

```
'use strict';

var assert = require('assert'),
  db = require('../lib/db'),
  nocklib = require('../lib/nocklib'),
  should = require('should');

var exchangeData = {};

suite('database', function() {
  var insertedOrder;

  test('open should open database connection', function(done) {
```

```
    db.open(done);
  });
  test('insertOne should insert a transaction', function(done) {
    var ord = nocklib.generateRandomOrder(exchangeData);
    db.insertOne('transactions', ord, function(err, order) {
      should.not.exist(err);
      should.exist(order._id);
      insertedOrder = order;
      done();
    });
  });
  test('findOne should find a single transaction', function(done) {
    var id = insertedOrder._id;
    db.findOne('transactions', id, function(err, order) {
      should.not.exist(err);
      should.exist(order._id);
      order.price.should.eql(insertedOrder.price);
      order.volume.should.eql(insertedOrder.volume);
      done();
    });
  });
});
```

Popišme si hlavní funkce našeho kódu. V prvním testu otevíráme spojení s databází MongoDB. Pokud selže tento test, selžou také všechny následující testy. Tak se dozvíme, že je nutné připojit se k databázi.

Abychom mohli vložit objednávku do databáze, nejprve ji generujeme pomocí funkce `generateRandomOrder()` z kapitoly 2. Posléze ji vkládáme do databáze. Náš test spočívá v ověření existence pole `_id`:

```
should.exist(order._id);
```

Poznámka: Identifikátory objektů
Identifikátor objektu je pole, které generuje MongoDB interně pro každý dokument v kolekci. Je jedinečný, a to dokonce mezi více stroji. Užitečnou vlastností tohoto pole je, že obsahuje také datum vytvoření, takže pokud ho chcete zjistit, nemusíte vytvářet samostatné pole.

Naši vyhledávací funkci testujeme tak, že porovnáváme náhodně vygenerovanou objednávku s objednávkou, kterou jsme právě vložili do databáze. Vyhledávání bude úspěšné, pokud se budou hodnoty `price` a `volume` aktuální objednávky shodovat se stejnými hodnotami v právě vložené objednávce.

```
order.price.should.eql(insertedOrder.price);
order.volume.should.eql(insertedOrder.volume);
```

Poznámka: Pojmenování polí
Měli byste používat popisné názvy proměnných. MongoDB se však vyznačuje tím, že opakuje názvy polí v nekomprimované podobě ve všech dokumentech. To znamená, že kdybyste měli milion záznamů zákazníka s polem pojmenovaným `isCustomerEligibleForDiscount`, tento databázový systém by ho zopakoval milionkrát. Protože nekomprimuje názvy polí, bývá zvykem volit krátké názvy polí pro kolekce, které mohou obsahovat velký počet dokumentů. Některé společnosti používají dokonce jen jednopísmenné nebo dvoupísmenné názvy polí.

Tvorba obalu

S nativním ovladačem může kód aplikace rychle narůstat. Proto si vytvoříme tenký obal okolo nativního ovladače, abychom abstrahovali některé nízkoúrovňové detaily. Nejprve importujeme některé moduly a spustíme sestavovací kód. Následující kód vložíme do souboru *lib/db.js*:

kapitola03/lib/db.js (*úryvek*)

```
var mongodb = require('mongodb'),
  Db = mongodb.Db,
  Server = mongodb.Server,
  DEFAULT_PORT = 27017;

var envHost = process.env['MONGO_NODE_DRIVER_HOST'],
  envPort = process.env['MONGO_NODE_DRIVER_PORT'],
  host = envHost != null ? envHost : 'localhost',
  port = envPort != null ? envPort : DEFAULT_PORT;

var db = new Db('nockmarket',
  new Server(host, port, {}),
  { native_parser: false });
```

Většinu tohoto sestavovacího kódu tvoří import a inicializace. Novou položkou je objekt `process`, jenž reprezentuje způsob, jak komunikovat s nativním prostředím. Může vykonávat nejrůznější operace – například měnit adresář, zkoumat využití paměti a určovat dobu od spuštění.[6]

V tomto případě pomocí něho prozkoumáváme proměnné prostředí – konkrétně jméno hostitele a port. Praxí osvědčeným postupem je ukládat konfigurační údaje (jako jsou kupříkladu jméno hostitele, port, uživatelské jméno a heslo) vně zdrojového kódu.

Představte si, že byste měli produkční verzi systému a chtěli v ní změnit port databáze. Kdybyste ho měli uložený ve zdrojovém kódu, museli byste měnit zdrojový kód a znovu zavést celý systém. V mezičase by ale některý vývojář nedodržel důkladně návrh řízený testy a zavedl vám do systému chybu. Vy tuto chybu zavedete do provozu jen kvůli tomu, že jste chtěli změnit port.

Vývojáři obvykle ukládají konfigurační údaje pomocí některé z následujících možností:

6 *http://www.nodejs.org/api/process.html*

- textových konfiguračních souborů,
- databázových úložišť,
- proměnných prostředí.

Ve všech těchto případech lze měnit konfiguraci bez zásahu do zdrojového kódu. V tomto příkladu ignorujeme proměnné prostředí, a tudíž hostitelem bude výchozí `localhost` a portem bude výchozí port.

Vyhledávání dat

Dále napíšeme několik funkcí pro prohledávání kolekcí a přidáme je do našeho kódu.

kapitola03/lib/db.js (*úryvek*)

```
module.exports = {
  find: function(name, query, limit, callback) {
    db.collection(name).find(query)
      .sort({ _id: -1 })
      .limit(limit)
      .toArray(callback);
  },
  findOne: function(name, query, callback) {
    db.collection(name).findOne(query, callback);
  },
  ...
```

Zde implementujeme dvě vyhledávací funkce. První z nich hledá všechny dokumenty, které odpovídají danému dotazu. Příkazem `db.collection(name)` vybíráme příslušnou kolekci. Kolekce je stejná jako tabulka v terminologii relačních databází. Funkcí `find()` spouštíme definovaný dotaz. Do podrobností o dotazovacím jazyku zabrousíme o něco později.

Pravděpodobně rádi uslyšíte, že syntaxe dotazovacího jazyka MongoDB vychází z JavaScriptu, takže máte na vaší cestě k mistrovství o jednu překážku méně.

Po spuštění daného dotazu řadíme výsledky v obráceném pořadí podle pole `_id`:

```
.sort({ _id: -1 })
```

Díky tomu se nejnovější položky budou nacházet na prvních místech. Následně aplikujeme omezení, aby nám databázový systém nevrátil obsah celé databáze:

```
.limit(limit)
```

A nakonec převádíme výsledek na pole a voláme funkci zpětného volání:

```
.toArray(callback);
```

Všimněte si, že volání metody `limit()` bezprostředně následuje za voláním metody `sort()` a to zase následuje za voláním metody `find()`. Tato technika se označuje jako **řetězení**. Jedná se o elegantní řešení, jak zkombinovat nejrůznější operace.

Funkce `findOne()` je téměř stejná jako funkce `find()`, ale s tou výjimkou, že vrací jediný výsledek – je jednoduší, protože v ní neřadíme, neomezujeme počet výsledků ani je nepřevádíme na pole. Většinou slouží pro vyhledávání podle hodnoty `_id`, což zaručuje, že najde jediný výsledek. Pokud vyhledávání probíhá nad jiným polem a databázový systém najde více výsledků, vrátí pouze první z nich.

Vkládání dat

Nyní se přesuňme k funkcím pro vkládání.

kapitola03/lib/db.js (*úryvek*)

```
insert: function(name, items, callback) {
  db.collection(name).insert(items, function(err, result) {
    callback(err, result.ops);
  });
},
insertOne: function(name, item, callback) {
  module.exports.insert(name, item, function(err, items) {
    callback(err, items[0]);
  });
},
```

Metoda `insert()` přijímá pole položek a vkládá je do databáze. Toto pole se skládá z objektů, které můžou mít libovolný tvar a nemají žádnou souvislost s předchozími nebo následujícími objekty. To jen poukazuje na povahu databází bez schématu. Jestliže chcete vložit finanční transakci, za ní záznam zaměstnance a potom recept na kuřecí polévku, nic vám nebrání (alespoň z technického hlediska ne). MongoDB přesouvá pravidla, která zabraňují takovému návrhu, z databázového systému do samotné aplikace.

Funkce `insertOne()` představuje tenký obal nad funkcí `insert()` – jednoduše volá tuto funkci a vrací první položku. Jedná se o **syntaktický cukr** – tímto termínem vývojáři označují kód, který se funkčně podobá jiné části kódu, ale je napsaný stručněji a elegantněji.

Teoreticky je funkce `insertOne()` zbytečná, protože bychom mohli vkládat a načítat jedinou položku tak, že bychom používali pole s jedinou položkou. K výsledku operace vkládání bychom tudíž přistupovali například takto: `result[0]`. Bylo by však lepší pracovat přímo s výsledkem, protože víme, že se bude jednat o jedinou položku. Může se to zdát malicherné, ale rozvážné používání syntaktického cukru v kódu vylepšuje čitelnost a udržovatelnost kódu. Na konec přidáme malou funkci, která otevírá spojení s databází:

kapitola03/lib/db.js (*úryvek*)

```
  ...
  open: function(callback) {
    db.open(callback);
  }
};
```

Naše testy jednotek jsou připravené ke spuštění. Přidáme následující závislosti do souboru *package.json*:

kapitola03/package.json (*úryvek*)

```
{
  ...
  "dependencies": {
    ...
    "ejs": "2.4.1",
    "express": "4.13.1",
    "mongodb": "2.1.14"
  }
}
```

Až provedete příkaz `npm install` a poběží vám démon MongoDB, můžete zadat příkaz `make test`, přičemž všechny testy jednotek by měly projít.

Ukládání dat

Jelikož jsme postavili lešení, můžeme upravit soubor *nockmarket.js* tak, aby ukládal objednávky do databáze – zatím je jen vypisuje do konzole. V prvé řadě se musíme připojit k databázi. Přidáme příkaz `db = require('./lib/db')` do seznamu importů a nahradíme volání funkce `submitRandomOrder()` následujícím kódem:

kapitola03/nockmarket.js (*úryvek*)

```
db.open(function() {
  submitRandomOrder();
});
```

Důležitým aspektem výše uvedeného kódu je, že má asynchronní povahu. Musíme totiž zajistit, že budeme připojeni k databázi, než začneme ukládat objednávky.

Poznámka: Synchronní kód

Stejná část kódu by v synchronním zápisu vypadala takto:

```
db.open();
submitRandomOrder();
```

Ve většině programovacích jazyků se jedná o běžný způsob zacházení se souvisejícími funkcemi. V JavaScriptu by to ale skončilo pokusem o odeslání objednávky před připojením k databázi, jelikož metoda `open()` není blokující.

Teď upravíme naši odesílací funkci následovně:

kapitola03/nockmarket.js (*úryvek*)

```
function submitRandomOrder() {
  var order = nocklib.generateRandomOrder(exchangeData);
  console.log('order', order);
  if (order.type == exch.BUY) {
```

```
    exchangeData = exch.buy(order.price, order.volume, exchangeData);
  } else {
    exchangeData = exch.sell(order.price, order.volume, exchangeData);
  }

  db.insertOne('transactions', order, function(err, ord) {
    if (exchangeData.trades && exchangeData.trades.length > 0) {
      var trades = exchangeData.trades.map(function(trade) {
        trade.init = (order.type == exch.BUY) ? 'b' : 's';
        return trade;
      });
      db.insert('transactions', trades, function(err, trades) {
        pauseThenTrade();
      });
    } else {
      pauseThenTrade();
    }
  });

  function pauseThenTrade() {
    var pause = Math.floor(Math.random() * timeRange) + timeFloor;
    setTimeout(submitRandomOrder, pause);
    console.log(exch.getDisplay(exchangeData));
  }
}
```

V prvním novém kusu kódu, ve volání metody `db.insertOne()`, vkládáme naši náhodně vygenerovanou objednávku do databáze. Jakmile ji vložíme a databáze nám vrátí výsledek, ověříme, jestli se uskutečnily nějaké obchody:

```
if (exchangeData.trades && exchangeData.trades.length > 0) {
```

Pokud se uskutečnily, doplníme pomocí vestavěné metody `map()` příznak, zda obchod zahájil kupující nebo prodejce:

```
var trades = exchangeData.trades.map(function(trade) {
  trade.init = (order.type == exch.BUY) ? 'b' : 's';
  return trade;
});
```

Tento krok děláme ze dvou důvodů. Chceme ukázat, jak MongoDB umí nakládat s různými typy dokumentů. Zavedení nového sloupce by představovalo komplikaci v relačních databázích. Museli bychom buď data rozdělit do více tabulek, nebo mít jedinou tabulku a v ní hodnoty `null` pro sloupce, které pro daný záznam nepotřebujeme. Ani jedno řešení není ideální. V MongoDB tuto situaci nemusíme vůbec řešit – můžeme jednoduše uložit různé typy dokumentů do stejné kolekce.

Druhým důvodem pro zavedení tohoto kódu je, že ukazuje, jak funguje metoda `map()`. Cílem mapování je transformovat současnou kolekci nějakým způsobem. V tomto případě používáme vestavěnou metodu `map()` jazyka JavaScript, které předáváme transformační funkci, v níž říkáme, jak změnit jednotlivé položky v kolekci.

Poznámka: MapReduce

Mapování je součástí velmi mocného paradigmatu MapReduce. Není nikterak přehnané tvrdit, že toto paradigma tvoří srdce softwarové infrastruktury společnosti Google. Společnost Google popisuje na svém oficiálním blogu, jak používá MapReduce k řazení jednoho petabajtu dat.[7]

Pokud nemáte určité zkušenosti s programováním, nemusí vám být koncepce mapování okamžitě jasná. Následuje příklad, jenž vám pomůže osvětlit tuto koncepci. Spusťte cyklus REPL příkazem **`node`** a potom napište tento řádek kódu:

```
[1, 2, 3].map(function(x) { return x * 2; });
```

Mělo by se objevit:

```
[ 2, 4, 6 ]
```

V tomto kódu berete vstupní pole a aplikujete na něj funkci, která násobí všechny položky pole dvěma. Jestliže si chcete ověřit, jak je mapování užitečné, zkuste přepsat kód bez metody **`map()`** tak, abyste zachovali jeho funkčnost. Jistě oceníte, o jak elegantní řešení na transformační úlohy se jedná.

Stav před frameworkem Express

Ačkoliv pro naše webové aplikace používáme primárně framework Express, byla by chyba nepobavit se krátce také o nativním modulu `http`. V rámci vedlejšího projektu postavíme vzdálený autentizační modul. Jeho hlavní funkci umístíme fyzicky na vzdálený server, ale budeme ji moct volat lokálně. To je možné díky modulu `dnode`, který vytvořil James Halliday.

Nejprve přidejte následující závislost do souboru *package.json*: `"dnode": "1.2.2"`.

Nyní spusťte příkaz `npm install` a vytvořte adresář s názvem *dnode*. V něm pak vytvořte soubor *auth.js*:

kapitola03/dnode/auth.js

```
var dnode = require('dnode');
dnode(function(remote, conn) {
  this.auth = function(user, pass, cb) {
    var users = {
      foo: 'bar'
    };
    var p = users[user];
    if (p === pass) {
      cb(null, 'PŘIHLÁŠEN!');
    } else {
```

7 *http://googleblog.blogspot.com.au/2008/11/sorting-1pb-with-mapreduce.html*