SAMPLE CHAPTER

Grails IN ACTION

SECOND EDITION

Glen Smith
Peter Ledbrook

FOREWORD BY Dierk König

MANNING

Grails in Action, Second Edition

by Glen Smith
Peter Ledbrook

Chapter 16

brief contents

16 NoSQL and Grails

This chapter covers

- Why NoSQL is worth considering
- How GORM interacts with NoSQL stores
- Key/value stores with Redis
- Document-oriented data storage with MongoDB
- Graph databases with Neo4J

In chapter 15, you worked on events, messaging, and scheduling tasks. In this chapter we take you on a tour of all the popular Grails NoSQL solutions, and give you a good sense of what makes sense where. You also learn how your relational and NoSQL data can happily co-exist (remembering that NoSQLstands for Not Only SQL). By the end of this chapter, you'll understand the brave and shiny new world of NoSQL solutions and know which ones are worth exploring in your next enterprise project.

16.1 The problem with PostgreSQL (or when to choose NoSQL)

You've used GORM with our favorite relational databases throughout the entire book, so why bother to look elsewhere? We hear you. There's a huge amount to be said for sticking with workhorse technologies that are battle-tested and proven in

the real world. Relational technology has been with us since Codd and Date's work in the 1960s, and we've all had professional experience with modern, fast relational databases. There's nothing wrong with using PostgreSQL—we highly recommend it if it suits your data storage needs. And then there's the catch…

If you haven't hit them already in your professional life, situations occur where you're willing to make a trade-off between the standard relational ACID model (and its normalized transactional goodness) and something that offers either one of the following:

- Much greater performance on the kinds of large datasets you deal with
- Much more flexible and extensible data structures than your typical normalized relational tables will permit

If you've ever tried to shoehorn a document-based structure into a relational model (where each document may have its own unique set of properties), you've already felt the pain of fighting city hall. In these scenarios, using a document-oriented database that supports these structures out of the box (such as MongoDB) makes sense. Cleaner code, cleaner data, better performance. What's not to love?

Similarly if you've ever tried to cram a tree or graph structure (such as a directory tree) into the relational model, it's a world of complex joins and indirections and various parent/child columns, and it becomes difficult to overcome the impedance mismatch. In those scenarios, having a database that "speaks graph natively" (such as Neo4j) is what you want. Traversing the tree is fast, the structures are logical and easy to move around, and no dodgy abstractions are required.

What kinds of data-storage operations does NoSQL bring to the table? In this section we take you on a rock-star tour of the finest tools in the NoSQL market.

16.2 Types of NoSQL databases (and typical use cases table)

Before we dive into the most common NoSQL options, let's survey popular products in the NoSQL world and the types of data that they excel at storing (see table 16.1). If nothing else, it's worth noting the terminology used for any NoSQL explorations you do down the road.

Table 16.1 NoSQL databases and types of data they contain

Type of NoSQL database	Common products	Typical examples of data stored
Key-value store	▪ Redis ▪ Memcached ▪ Voldemort ▪ Basho Technologies Riak ▪ Tokyo Tyrant	Persistent hash tables, session tokens, global state, counters (such as API meters)
Document-oriented store	▪ MongoDB ▪ Apache CouchDB ▪ Apache Jackrabbit ▪ Elasticsearch	User profile data (with free-form fields), survey and questionnaire data, "objects" with their properties

Table 16.1 NoSQL databases and types of data they contain *(continued)*

Type of NoSQL database	Common products	Typical examples of data stored
Graph database	▪ Neo Technology Inc. Neo4j ▪ Orient Technologies OrientDB	Social network graphs (Facebook graphs), directory and tree structures, query link depth on related data
Column database	▪ Apache HBase ▪ Apache Cassandra ▪ Google Bigtable	Time series data

Now that you've seen the common types of NoSQL solutions, it's time to explore the field with one of the most common NoSQL services deployed today: a key/value server called Redis.

16.3 Using Redis to work with key-value stores

You might think of a key-value NoSQL store as a massive persistent hash table: you send it your key/value pairs to hold, and you pull back your values by key later on. The advantages of doing this in a NoSQL store rather than in your own app is that you don't have to worry about

- The app-server crashing/restarting
- Writing anything to disk to save your hash state
- Configuring any kind of persistent caching solution (such as the ones we looked at in chapter 15)
- Providing an API for other applications to share your hash table data

One extremely popular hash table for these kinds of operations is Memcached. It's lightweight, easy to set up and replicate, and lightning fast because it stores data only in memory. It's also common on cloud services and runs great in both Windows and UNIX-like environments. The only snag is that it isn't persistent, so your data never survives a restart.

Then Redis came along—a better Memcached than Memcached! It offers all the benefits of Memcached (memory-based, single-threaded, lightning fast), but adds a truckload of compelling features to boot (persistence, master/slave replication, lists, sets, queues and unions, and all with transactionality). Being backed by a big vendor (VMware), it also feels like it's going to be stable, supported, enhanced, and around for while!

Tons of high-traffic sites use Redis (think GitHub, Digg, Stack Overflow, and Disqus to name a few), and the technology is rock solid. Thanks to its popularity, you also find it deployed on all the popular cloud environments (Heroku, OpenShift, and Cloud Foundry)

16.3.1 Installing your own Redis server

If you're not running your app on one of the existing cloud services (and chances are you won't be for your first iteration), you need to install a local copy of Redis to test

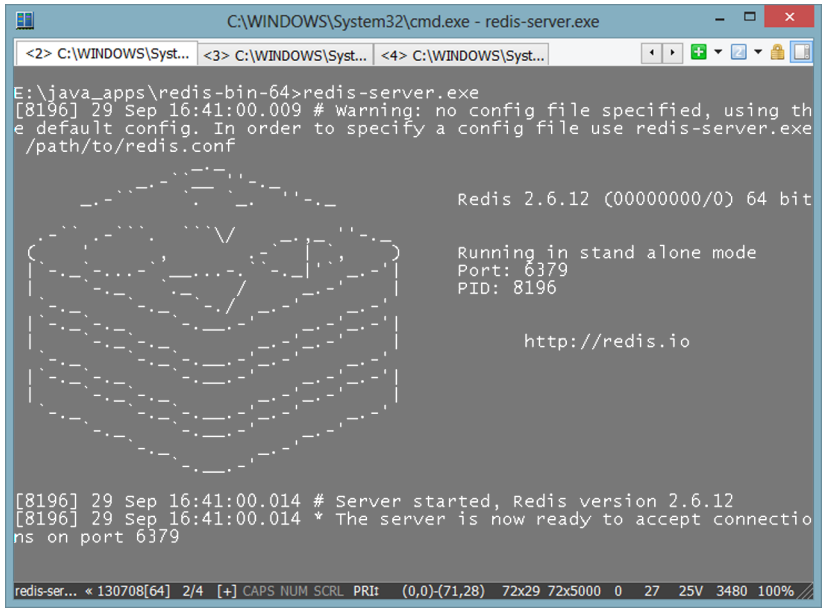


Figure 16.1 The Redis welcome screen

against. Fortunately, it's easy to install and available for all your favorite OSes. Head over to www.redis.io/ to download a copy for your target OS. If you're on Windows (which you will be in this chapter), grab the Microsoft version of Redis that you find on its GitHub site. You want redisbin.zip (32-bit Windows) or redisbin64.zip (64-bit Windows), which you'll find at https://github.com/MSOpenTech/redis/tree/2.6/bin/release.

Windows users can simply unzip the binary and they're ready to roll. No further setup is required. Linux and OS X users will need to Untar/zip their distro of choice, then run a `make` to place a compiled Redis binary in `./src` subdirectory. You can start up your local Redis by running the `redis-server` executable in your installation directory. You're greeted with the famous Redis ASCII art shown in figure 16.1.

With your Redis server installed and started, it's time to learn a few of the basic Redis commands using the command-line client.

16.3.2 Using Redis operations

With Redis installed, the best way to experiment with the service is to use the `redis-cli` executable found in the same directory where you unzipped the service. Start it up and experiment with the common Redis commands listed in table 16.2. Running these yourself will acquaint you with the commands you'll use via the Redis Grails API in the next section.

Table 16.2 Redis commands and descriptions

Command samples	Description

`set name glen`
`get name`
`exists glen`
	Placing a value in the cache, reading it's current value, and testing for it's existence

`incr hitcount` `decr hitcount`	Incrementing and decrementing a counter

`rpush users glen`
`lpush users peter`
`lrange users 0 1`
`lpop users`
`rpop users`
	Lists can be pushed, popped, and inspected from either left or right side

`hset email glen glen@bytecode.com.au`
`hget email glen`
`hvals email`
`hkeys email`
	Hashtables are supported natively

`sadd fruit orange`
`smembers fruit`
`sinter fruit citrus`
`sdiff fruit citrus`
`sunion fruit citrus`
	Sets ensure uniqueness and support common set operations such as intersection, diff, and union

As you can see, Redis goes beyond caching and returning simple values. It also has first-class support for atomic integer operations (perfect for counters), lists, hashes, and sets. You explore more of that goodness in the next few sections.

Diving deeper into Redis

To learn more about all the available Redis commands, we recommend you check out the excellent online reference at http://redis.io/commands. If you want to see everything that Redis via Groovy has to offer, we highly recommend all the amazing Redis presentations given by Groovy great Ted Naleid; they're linked off the Grails Redis plugin page at http://grails.org/plugin/redis.

16.3.3 Installing the Redis plugin (including pooling configuration)

The first question that you need to ask is "which Grails Redis plugin?" because two are available:

1. *Grails Redis plugin*—Provides a nice wrapper for the underlying Jedis library, with much Grails goodness baked in (taglibs, Grails service, annotations, and so on). See http://grails.org/plugin/redis.
2. *Grails Redis GORM Plugin*—Offers GORM support for Redis, including the ability to store Grails domain objects in Redis and use the standard GORM goodness

you're used to, including dynamic finders, criteria queries, named queries, and so on. See http://grails.org/plugin/redis-gorm.

To be honest, the low-level API of the Redis plugin makes more sense for what you need because Redis isn't the ideal place to store and query GORM objects (and the base Redis plugin offers better Grails support for all the common places you typically want to interact with Redis from Grails).

Let's add the latest Redis plugin to your /grails-app/conf/BuildConfig.groovy, so you'll be ready to cache up a storm.

```
plugins {
    ...
    compile ":redis:1.3.3"
    ...
}
```

With the plugin installed, your application is equipped with a range of Redis enhancements including

- A `redisService` Spring bean that wraps all the low-level Redis API, as well as many Grails-specific convenience methods.
- A `redisPool` Spring bean that gives you low-level access to a pool of Redis connections (though typically you let `redisService` transparently handle all pooling for you).
- A `redis:memoize` taglib that lets you cache sections of your GSP pages (with timeout).
- A series of Redis-backed annotations, such as `@Memoize`, `@MemoizeList`, `@MemoizeHash`, `@MemoizeDomainObject`, `@MemoizeDomainList`, that return a cached object (or fetch the object and cache it if required).

Do you notice all the "`@Memoize`"ing happening around here? Perhaps we'd better introduce you to the simple meaning behind this complex term.

16.3.4 Simple, expiring key/value caching: what is all this @Memoize stuff?

One of the most common ways to use Redis is to cache your expensive data values and calculations. The quickest and easiest way to do this in Grails is using the range of handy annotations that ship with the Redis plugin. They all have an `@Memoize` prefix, which might be a new term for you, but don't be scared off, a quick example clears things up.

If you haven't yet come across this computer science term, it's a concise way to refer to an optimization technique you've probably already seen (and used) when you looked at the Cache plugin in chapter 10. Consider something such as this:

```
@Memoize(key = "#{user.loginId}", expire = "60000")
def performExpensiveUserProfileOperation(User user) {
    log.info "${user.loginId} not in cache,
                ➥ performing expensive calculation"
    return user.doSomeExpensiveOperation()
}
```

The first time the `perfomExpensiveUserProfileOperation` method is invoked with a given user, the expensive calculation is performed and the value is cached in Redis based on the user's `loginId`. The second time the method is called with the same user as an input, the cached value is immediately returned (thereby skipping the expensive calculation).

The `expire` parameter specifies that you want this value cached for up to 60 seconds only (the value is specified in milliseconds), and after that you want to expire the value from the cache and recalculate it. You might hear this called the TTL (time-to-live) of the value.

Under the covers, the Redis plugin looks at all the interactions with the back end to set and get the keys based on the user's `loginid`, but the flexibility of the annotation allows you to use whatever keys you like.

The `@Memoize` annotation is perfect for all your service classes that perform data lookups and calculations. But what if you need to cache in the view tier? Well, the Redis plugin provides the taglib you need.

16.3.5 Working with the Redis taglib

When we introduced the Cache plugin in chapter 10, we showed you the `cache:cache` taglib, which allows you to cache a portion of a GSP into an in-memory cache for later reuse. Remember this old chestnut for caching the user count?

```
<cache:block>
    Hubbub currently has ${ com.grailsinaction.User.count()} registered users.
</cache:block>
```

In this case you don't specify a timeout because you configured the underlying cache with a set timeout value.

The Redis plugin gives you the same capability, but this time it's backed by a Redis store, so you can safely use it in clusters, and it happily survives restarts. It comes as no surprise that the Redis plugin embraces a similar semantic:

```
<redis:memoize key = "hubbubCount", expire = "60000">
    Hubbub currently has ${ com.grailsinaction.User.count()} registered users.
</redis:memoize>
```

In this case you need to provide a key to use in the Redis store, and, optionally, an expire value, otherwise the value lives forever.

Backing the standard Grails Cache with Redis

You may wonder whether it's possible to use Redis to back the standard Grails Cache plugin. The answer is yes! A special Grails plugin called the Grails Redis Cache plugin (http://grails.org/plugin/cache-redis) plugs into the existing Grails cache beans and backs them with Redis. Check out the plugin page for more details.

16.3.6 Beyond the basics: working with the Redis service object directly

We showed you the most common use cases for interacting with key/value stores such as Redis—storing and retrieving expiring values in a persistent cache. But when we introduced Redis, we said it could do more than that. We talked about sets, lists, hashes, and atomic integers. It's time to unleash all that power. The Redis plugin provides you with the lower-level `redisService` Spring bean for such operations.

For your Redis enhancements to Hubbub, we're going to introduce a `StatsService` object responsible for keeping various stats on Hubbub's operation (such as the number of posts made today and the highest posting users).

Let's get your scaffolding in place for your service object with your familiar `grails create-service com.grailsinaction.Stats`. For your stats object, it would be nice to hook into any new Post objects created in the system, which should make you think about the Platform Core Events capabilities we introduced in chapter 15.

Let's hook your `StatsService` into any newly created posts and keep a cache of the number of posts created today, as shown in the following listing.

Listing 16.1 A `StatsService` storing daily totals in Redis

```
package com.grailsinaction

class StatsService {

    static transactional = false

    def redisService

    @grails.events.Listener
    void onNewPost(Post newPost){

        String dateToday = new Date().format("yy-MM-dd")
        String redisTotalsKey = "daily.stat.totalPosts.${dateToday}"

        log.debug "New Post from: ${newPost.user.loginId}"

        redisService.incr(redisTotalsKey)

        log.debug "Total Posts at: ${redisService.get(redisTotalsKey)}"

    }
}
```

❶ Injects redisService for Redis integration

❷ Works out daily unique Redis key for caching

❸ Increments post count for today

❹ Logs out current totals

To take advantage of standard Grails injection, you inject the `redisService` ❶ that the plugin provides. Use this service object to invoke any of the standard Redis operations included in the Redis command reference we introduced you to previously (http://redis.io/commands. See the following sidebar for more information on how this service works under the hood).

With your service acquired, you need to work out which key to use in Redis to store your daily post counts. Let's use the current date as a key ❷ with `yy-MM-dd` qualifiers, for example, daily.stat.totalPosts.13-09-30 indicates September 30, 2013.

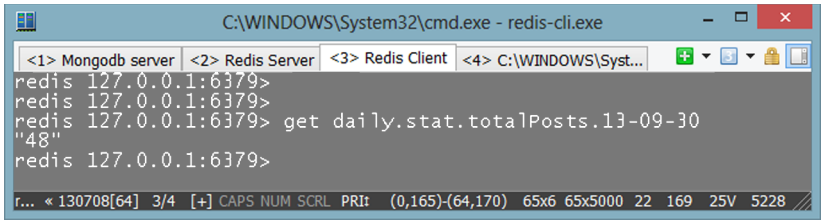


Figure 16.2 After you create a few posts in Hubbub, you can use the Redis client to get your current values.

With a Redis service handle and your key calculated, incrementing the posts for the day is a simple matter of invoking the `incr` method ❸. This is atomic and inherently thread-safe (because all Redis servers are single-threaded).

For fun, you log out the current value of this counter ❹ to make sure your changes stick. You could instead use your command-line Redis client, as shown in figure 16.2, to inspect the value.

Wow, 48 posts already. It's been a good day of testing.

> **Why doesn't my IDE autocomplete redisService methods?**
>
> You may wonder why your IDE isn't autocompleting the various methods on the `redisService` such as `incr`. The reason is that the plugin is implemented using Groovy's `methodMissing` metaclass feature (the same techniques that GORM uses for dynamic finders).
>
> The plugin literally catches any method you invoke on the service, then looks for a matching method in the underlying Jedis Java library. It's easier to enhance the plugin with new features that appear in the underlying library but harder for you to find out exactly which method you need to call!

It's nice to have your daily stats in place, but it would be even nicer if you could break them down by top posters of the day. And that sounds like a perfect way to introduce you to one of Redis's most powerful data structures: the sorted set.

16.3.7 Top posters with Redis sorted sets

In your Redis explorations you looked at caching simple values (such as strings), and worked with the atomic counter support offered by `incr` (and `decr`). But one of Redis's most powerful features is its support for lists, sets, and hashes. Users have even described Redis as "a collection of data structures exposed over the network."

If you had a "Top Posters of the Day" feature, you'd need to keep a set of user IDs and their counts. You can easily do that in a hash, but Redis offers first-class support for these kinds of counting tables through sorted sets. Every entry in a Redis sorted set records a name and a score. You can then efficiently perform various operations on the order sets, such as retrieving the list ordered by score (in either direction), retrieving

values higher/lower than a given score, finding the score for a given entry, incrementing and decrementing scores for a given name, and so on.

In your Post of the Day sample, you take advantage of these sorted sets to keep a `loginId` along with a count of posts for the day. To keep it simple, you key the whole ordered set off the current date (as you did for the daily totals stat).

To increase the score of an element in a sorted set, use the Redis command `zincrby` (the `Z` is used to prefix all sorted set operations). To invoke the command (and the Grails method), use the following arguments:

```
ZINCRBY cacheKey incrBy name
```

To increment the count of posts for `chuck_norris` by 1 on September 30, 2013, you issue this command:

```
redisService.zincrby("daily.stat.totalPosts.13-09-30", 1, "chuck_norris")
```

To retrieve the current value of that element, use the `ZSCORE` method:

```
redisService.zscore("daily.stat.totalPosts.13-09-30", "chuck_norris")
```

With a basic knowledge of sorted sets under your belt, let's dive into the implementation, as shown in the following listing.

Listing 16.2 A `StatsService` with total posts by user per day

```
package com.grailsinaction

class StatsService {

    static transactional = false

    def redisService                                          // 1 Injects redisService for Redis integration

    @grails.events.Listener
    void onNewPost(Post newPost) {

        String dateToday = new Date().format("yy-MM-dd")
        String redisTotalsKey = "daily.stat.totalPosts.${dateToday}"

        redisService.incr(redisTotalsKey)

        String redisTotalsByUserKey = "daily.stat.totalsByUser.${dateToday}"

        redisService.zincrby(redisTotalsByUserKey,
                             1, newPost.user.loginId)
        int usersPostsToday = redisService.zscore(redisTotalsByUserKey,
     newPost.user.loginId)
        log.debug "Incremented daily stat for ${newPost.user.loginId} to
     ${usersPostsToday}"

    }
}
```

// 2 Works out daily unique Redis key for caching daily sorted set

// 3 Increments post count for user's daily tally

// 4 Fetches current daily post count for user

Once again, you take advantage of your injected `redisService` ❶ to do all the low-level work. You calculate your key using the current date with namespacing ❷, then

you get to work incrementing the count for the current user ❸. If the user doesn't have a current value in the set, Redis assumes the current value is zero and increments it to one.

To prove to yourself the value is persisting, you fetch the current value for the user ❹ and log it out. This isn't necessary in prod code. In the real scenario, you'd fetch those values; you can use the `zrevrangeWithScores` method to fetch back a list in reverse sorted order (highest to lowest). Jedis returns these as an ordered list of tuples, in which you can get at each element using the `element()` and `score()` methods with code similar to the following listing.

Listing 16.3 Getting back an ordered list of top posters for the day

```
def getTodaysTopPosters() {
        String dateToday = new Date().format("yy-MM-dd")
        String redisTotalsByUserKey = "daily.stat.totalsByUser.${dateToday}"
        def tuples = redisService.zrevrangeWithScores(
                            redisTotalsByUserKey, 0, 1000)
                                    tuples.each { tuple ->
            log.debug("Posts for ${tuple.element} -> ${tuple.score}")
        }
        return tuples
    }
```

❶ Fetches ordered list of top posters (highest to lowest)

❷ Iterates list outputting name and score

Using your `redisService`, you grab and reverse-order the list from your sorted set (reverse in the sense that it's ordered highest to lowest). This routine takes two arguments ❶: the first is the minimum count to retrieve. Because no one gets into this set without at least one post, you set this value to zero. The second is the maximum score to retrieve (which you set to 1,000 as an arbitrarily high value).

In this case you log out the results ❷, which sends the list to your console:

```
DEBUG grailsinaction.StatsService  - Posts for frankie -> 12.0
DEBUG grailsinaction.StatsService  - Posts for phil -> 10.0
DEBUG grailsinaction.StatsService  - Posts for graeme -> 4.0
```

And now that you're across common sorted set operations, this completes your whirlwind tour of Redis.

Additional Redis features to explore

We whetted your appetite for all the goodness available in Redis via Grails but recommend that you look through the online Redis plugin documentation on GitHub (https://github.com/grails-plugins/grails-redis), which has full coverage of all the available Redis annotations (and detailed configuration guides for pooling, pipelining, and other advanced features).

If you want to explore further how to integrate GORM with Redis, we recommend you also check out Grails's Redis GORM plugin (http://grails.org/plugin/redis-gorm) to see how you can augment domain classes for storage in Redis via GORM. It's a little fiddly at the moment, but many of the common GORM operations are well supported.

It's time to branch out into the document-oriented world of MongoDB.

16.4 Using MongoDB to work with document-oriented data

Your second set of NoSQL technologies to explore revolves around document-oriented NoSQL, in which the biggest player is MongoDB.

> **MongoDB: company and open-source community**
> MongoDB Inc. (formerly 10gen) builds and supports MongoDB, the open-source database (www.mongodb.org), and MongoDB Enterprise, the commercial edition of MongoDB (www.mongodb.com).

Document-oriented stores specialize in storing data items as self-contained objects rather than as key/value pairs. They provide fast ways to query and update these documents (and because they don't typically need to do any joins, the performance implications of querying documents can result in lightning-fast responses).

Aside from the potential speed improvements, what's so attractive about document-oriented databases and MongoDB in particular?

- *MongoDB is schemaless.* In a relational model, you have to decide your table structure and list of column names ahead of time, because refactoring columns can be a pain later. In MongoDB, every document can have its own custom set of fields, and you can change them (including adding and removing fields) whenever you like. That's amazing flexibility that you can let your database grow with your software.
- *MongoDB offers easy scalability options.* Scaling out to a cluster of relational databases can be tricky business. Deciding on sharing strategies remains a black art. MongoDB was designed with scalability in mind. Add more MongoDB servers to your config and MongoDB redistributes your documents for optimal load sharing and failover.
- *MongoDB is fast on Big Data.* MongoDB is short for "hu*mongo*us DB" and ships with a rich indexing model designed for storing gigantic datasets. Running MongoDB servers on commodity hardware is likely to give you more fast storage than you can ever use.
- *MongoDB offers native support for files and other large binary content.* Need to store pictures or other binary content in your database? MongoDB has built-in support for storing large files and their metadata.
- *Ubiquitous cloud services.* Most of the popular cloud operators (such as Heroku, OpenShift, and Cloud Foundry) all offer native MongoDB support on their cloud offerings. Companies such as MongoLab and MongoHQ offer "Mongo as a service" on a per-month, hosted-service basis.
- *Zero-cost kickoff, with great vendor support.* MongoDB is free under the Affero General Public License (AGPL) to use and run, but you can also purchase great

commercial support from MongoDB Inc. You don't have to skimp on big vendor backing if it's important to your scenario.

What do MongoDB documents look like? Let's work through an example, to give you an idea of what this document business is all about. Imagine you're storing a user's questionnaire results in your data store. Instead of a typical relational model (where you'd need to have a table with userid, question number, and question response), MongoDB stores the whole set of data in a single related document that it represents in a JSON format similar to the following code:

```
{
    "_id" : ObjectId("5248d92ae102251e9e94eb4b"),
    "title" : "q1",
    "question" : "What is your favourite colour",
    "answer" : "orange"
}
```

As you can see, this is the JSON that you know and love from your Ajax work in part 2 of the book, and which we dived even deeper into in chapter 13 when we looked at single-page web applications).

Internally, MongoDB stores documents in a special binary version of JSON known as BSON. But to the outside world, documents present as standard JSON that you can use with all the JSON tools you're used to. Because this is such a ubiquitous data format, developers are drawn to MongoDB because they can use their familiar tools and libraries.

But you aren't limited to "flat" documents of properties; you can also nest documents within one another as subdocuments. Imagine a blog system where you want to keep all the comments with their respective blog entry:

```
{
    title: "MongoDB rocks!",
    author: "Glen Smith",
    content: "I've been experimenting with MongoDB and it looks amazing",
    created:  ISODate("2013-09-30T14:00:00Z"),
    comments: [
        {
            comment: "Yeah, looks really promising",
            author: "Joe User",
            created: ISODate("2013-09-30T15:00:00Z")
        },
        {
            comment: "Cool. I must check it out",
            author: "Jill User",
            created: ISODate("2013-09-30T16:00:00Z")
        }
    ]
}
```

In addition to the ability to nest documents within documents and query them efficiently, you can structure your data in whatever way makes the most sense to your particular application without having to worry about any kind of schema definition up front.

We hope by now we piqued your interest in this bold, new document-oriented world. Let's take a small detour to introduce MongoDB terminology, then you'll create and query documents of your own.

16.4.1 Learning MongoDB terminology

In the relational world, you talk about tables and rows. But tables and rows don't make sense in a document-oriented world. Table 16.3 introduces the way MongoDB thinks about storage.

Table 16.3 MongoDB terminology

Relational database term	MongoDB equivalent
Database	Database
Table	Collection
Row	Document
Field	JSON property on a document
Primary key	Primary key
Index	Index

We'll walk you through working with collections and documents, but first, let's get all your tools and servers set up.

16.4.2 Getting set up: installing a MongoDB server

First you need to grab a MongoDB server for your platform of choice. It's a free download at www.mongodb.org/downloads.

After you unzip the installation, create a data directory to hold your MongoDB database data. By default, this directory is located at C:\data\db on Windows, and /data/db on UNIX and derivatives. You need to create this directory before you spark up MongoDB. Alternatively, you can tell MongoDB where your data directory is by passing in `--dbpath c:\my\custom\path`, but we assume you'll use the defaults.

To launch the MongoDB daemon, head into the bin directory of your unzipped MongoDB server, and run the `mongod` command as shown in figure 16.3.

It's time to fire up a client and connect to it.

16.4.3 Creating your first database

MongoDB ships with a command-line client called, well, `mongo`, which you'll find in the same bin directory as the MongoDB server. To create your first database, complete the following steps:

1. Start up the `mongo` client:

```
E:\java_apps\mongodb-win32-x86_64-2008plus-2.4.6\bin>mongo
MongoDB shell version: 2.4.6
connecting to: test
```

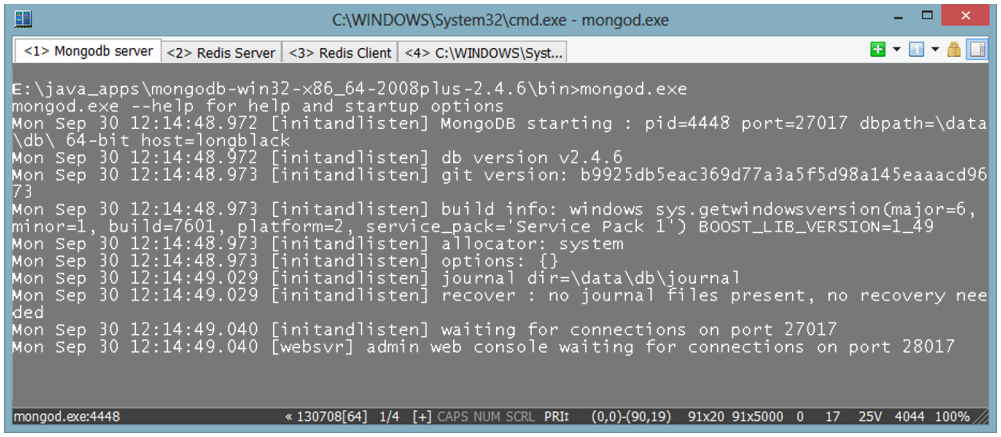


Figure 16.3 Launching the MongoDB server

2 Create a new database called "quiz" by switching to it with the `use` command:

```
> use quiz
switched to db quiz
```

MongoDB creates the database automatically when you switch to it. Once in a database, you create a `responses` collection to house all your response documents (remember, a collection is analogous to a table in relational parlance). Collections are automatically created when the first document is inserted into them.

3 Use the `insert` method on the `responses` collection to pass in your JSON objects, which represent each document:

```
> db.responses.insert({ title: "q1",
    ➥ question: "What is your fave color", answer: "orange" })
> db.responses.insert({ title: "q1",
    ➥ question: "What is your fave color", answer: "blue" })
> db.responses.insert({ title: "q1",
    ➥ question: "What is your fave color", answer: "green" })
```

With your collection populated with documents, you can give MongoDB queries to resolve, such as the count of documents in a collection:

```
> db.responses.count()
3
```

Or, if you're chasing the equivalent of a SELECT * FROM RESPONSES, you can display all the documents MongoDB has in a collection using the `find()` command:

```
> db.responses.find()
{ "_id" : ObjectId("5248e2dbb97b0d6acea283bb"),
    "title" : "q1", "question" : "What is your fave color",
    "answer" : "orange" }
```

```
{ "_id" : ObjectId("5248e2e6b97b0d6acea283bc"),
    "title" : "q1", "question" : "What is your fave color",
    "answer" : "blue" }
{ "_id" : ObjectId("5248e2edb97b0d6acea283bd"),
    "title" : "q1", "question" : "What is your fave color",
    "answer" : "green" }
```

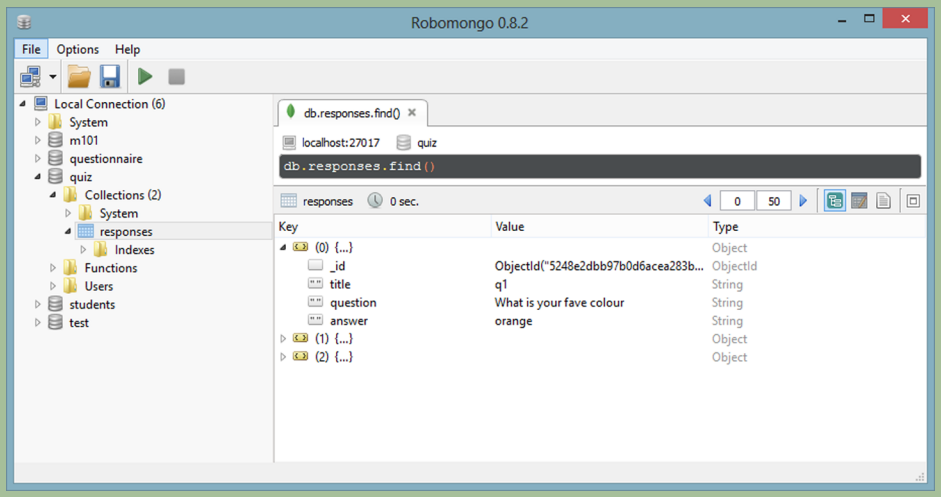
Notice that MongoDB automatically assigns an `ObjectId` element to the documents after they're inserted. This is a unique key for the object in the database.

What if you want to qualify your finds? No sweat. Pass in the argument(s) you want to constrain to your `find()` call, and MongoDB does the heavy lifting. Here's the equivalent of a SELECT FROM RESPONSES WHERE ANSWER='green':

```
> db.responses.find({answer: 'green'})
{ "_id" : ObjectId("5248e2edb97b0d6acea283bd"),
    "title" : "q1", "question" : "What is your fave color",
    "answer" : "green" }
```

Robomongo instead of the command line

If messing about on the command line seems tedious to you, we understand. Fortunately, you have a range of MongoDB GUIs and web apps that can make all this experimenting less painful. If a rich GUI takes your interest, we recommend you check out Robomongo (http://robomongo.org/). It's free and available on all the major platforms.



Robomongo is a GUI-based management tool for MongoDB.

Now that you've experimented with the MongoDB console, let's look at how to integrate Grails into this document-oriented world. Start by installing the MongoDB plugin.

16.4.4 Installing the MongoDB plugin

As you might imagine, the first step in making Hubbub MongoDB-ready is to install the Grails Mongo plugin (http://grails.org/plugin/mongodb). Add the latest MongoDB plugin to your /grails-app/conf/BuildConfig.groovy so you can get cracking on your Grails integration effort:

```
plugins {
    ...
    compile ":mongodb:1.3.0"
    ...
}
```

If you run MongoDB on your local host and the default port, no further configuration is required. If you connect to an external cloud-hosted solution, you can always customize your MongoDB server as shown in the following code:

```
grails {
  mongo {
        host = "yourRemoteService"
        port = 27017
        // or replicaSet = [ "server1:27017", "server2:27017"]

        databaseName = "hubbub"
    }
}
```

With the plugin installed (and optionally configured), it's time to point your domain classes toward MongoDB.

16.4.5 Polyglot persistence: Hibernate and MongoDB working together

If you use only MongoDB for your persistence engine, you can safely remove the Hibernate plugin entirely from your /grails-app/conf/BuildConfig.groovy. However, in your case, you're going to augment your existing Hibernate solution with new domain classes that are stored in MongoDB. This strategy is sometimes called *polyglot* persistence because you use several persistence engines in a single application.

One immediate candidate for your MongoDB integration is your `AuditService`. At the moment you log out your audit data to a file, but if you persist it in MongoDB, a range of query operations are available that you can expose to your admin users.

First up, in the following listing, let's create an `AuditEntry` domain object to hold your audit data.

Listing 16.4 Defining an `AuditEntry` object to store in MongoDB

```
package com.grailsinaction

import org.bson.types.ObjectId

class AuditEntry {

    static mapWith = "mongo"
```

(1) Tells Grails to use MongoDB for domain class.

```
    ObjectId id
    String message
    String userId
    Date dateCreated

    static constraints = {
        message blank: false
        userId blank: false
    }

    static mapping = {
        collection "logs"
        database "audit"
        userId index:true
        version false
    }
}
```

❷ Defines ObjectId to let Mongo assign its own IDs.

❸ Autotimestamping works fine for Mongo.

❹ Custom collection name (defaults to class name).

❺ Custom database name (defaults to app name).

❻ Indexes common query fields and increases performance.

❼ Turns off versioning.

This GORM domain class looks remarkably like any other GORM domain class. Aside from the `mapsWith` ❶ property (which is only required because you're getting Hibernate and MongoDB to coexist in one app), it consists of all the field definitions, autotimestamp fields ❸, constraints block, and indexing operations you know and love.

You'll notice, however, a few artifacts that are unique to how MongoDB-GORM interacts. You supplied an ID field ❷ declared as type `org.bson.types.ObjectId` (see the following sidebar). You also provided a custom mapping block to tune exactly where and how MongoDB stores this domain class. In this block you specified the MongoDB collection name ❹ and database name ❺, which, if not specified, defaults to the `classname` and `appname` respectively.

Even though you're in NoSQL land, for best performance you still need to index any fields that are likely to be common query candidates ❻, and because you won't ever edit or update an `AuditEntry`, you probably want to turn off versioning ❼.

What's with the ObjectId?

Normally you don't declare an ID field on Grails domain classes—you let Grails assign one for you. This causes a snag in MongoDB-land, however, because if you don't declare an ID, Grails stores an ordinal `Long` as the ID, which breaks MongoDB advantages such as autosharing. The best strategy is to declare it as an `org.bson.types.ObjectId` and let MongoDB assign an ID to that string when the object is stored (much like you saw in your console examples).

After all the plumbing is in place, persisting objects using the standard GORM APIs works. Here's your enhancement to the existing audit service to log all new post creations back to MongoDB:

```
@grails.events.Listener
    def onNewPost(Post newPost){
        log.error "New Post from: ${newPost.user.loginId} :
               ➥ ${newPost.shortContent}"
```

```
        def auditEntry = new AuditEntry(message: "New Post:
                ➥ ${newPost.shortContent}",
                userId: newPost.user.loginId)
        auditEntry.save(failOnError: true)
    }
```

With all the familiar `save()` semantics you're used to, you'd think this was heading into a relational data source if you didn't know the backstory. Let's run a few posts and see where these domain classes end up.

With a quick browse of the database in Robomongo, you can see your new entries persisting nicely in figure 16.4.

Notice the collection name is set to `logs` and the database name to `audit` as configured in the domain class (see listing 16.4). Also notice that MongoDB assigned an appropriate `_id` field on the document using its standard semantics.

Until now you've stored objects as you would with any relational back end. Now it's time to explore GORM's support for MongoDB's schemaless operations.

16.4.6 Stepping outside the schema with embeddables

One of the great advantages of document databases is storing all the data related to an object within a single document—typically through a kind of embedded subdocument. Fortunately the MongoDB GORM plugin knows all about this style of operation, so let's explore the support for these embedded documents.

One of the simplest forms of embedding is taking advantage of standard lists and maps on your domain classes. Imagine that when you store your `AuditEntry` domain classes, you want to dump out not only the name of the operation happening now, but also everything you know about the object under audit (such as all the properties of a newly created post). But you need that to be generic for all the kinds of objects you may audit in the future.

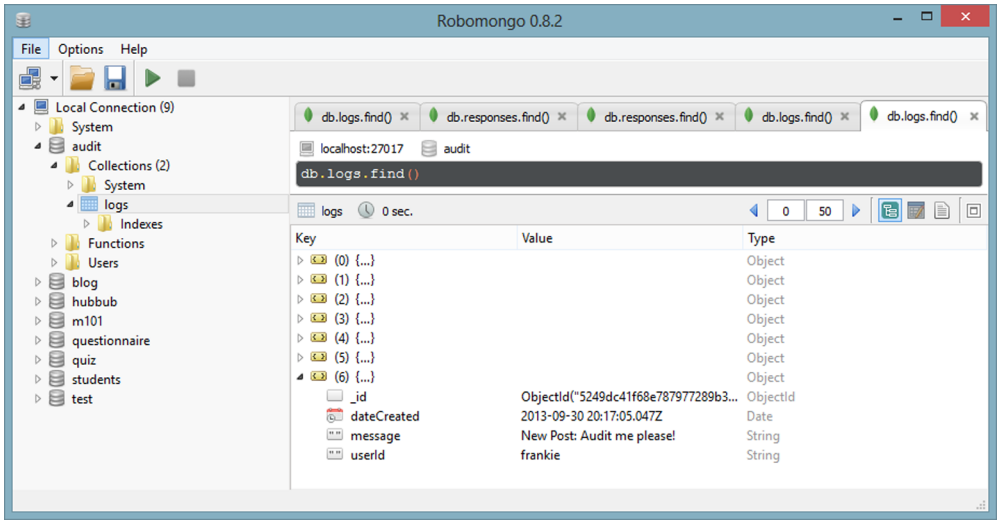


Figure 16.4 New entries in Robomongo

Let's capture the details of the object being audited in your `AuditEntry` via a `details` `Map`. Most of the domain class is omitted, but you'll get the gist:

```
class AuditEntry {
    ObjectId id
    // ...other fields omitted
    Map details
}
```

Now that you have your embedded `Map` object, it's a simple matter to dump out all the properties of the object under audit directly to that map. Something like a straight properties assignment should do the trick:

```
@grails.events.Listener
def onNewPost(Post newPost){
    def auditEntry = new AuditEntry(message: "New Post:
        ➥ ${newPost.shortContent}", userId: newPost.user.loginId)
    auditEntry.details = newPost.properties
    auditEntry.save(failOnError: true)
}
```

If you browse the next newly minted `AuditEntry`, you see your embedded `details` properties object as a subdocument, as shown in figure 16.5.

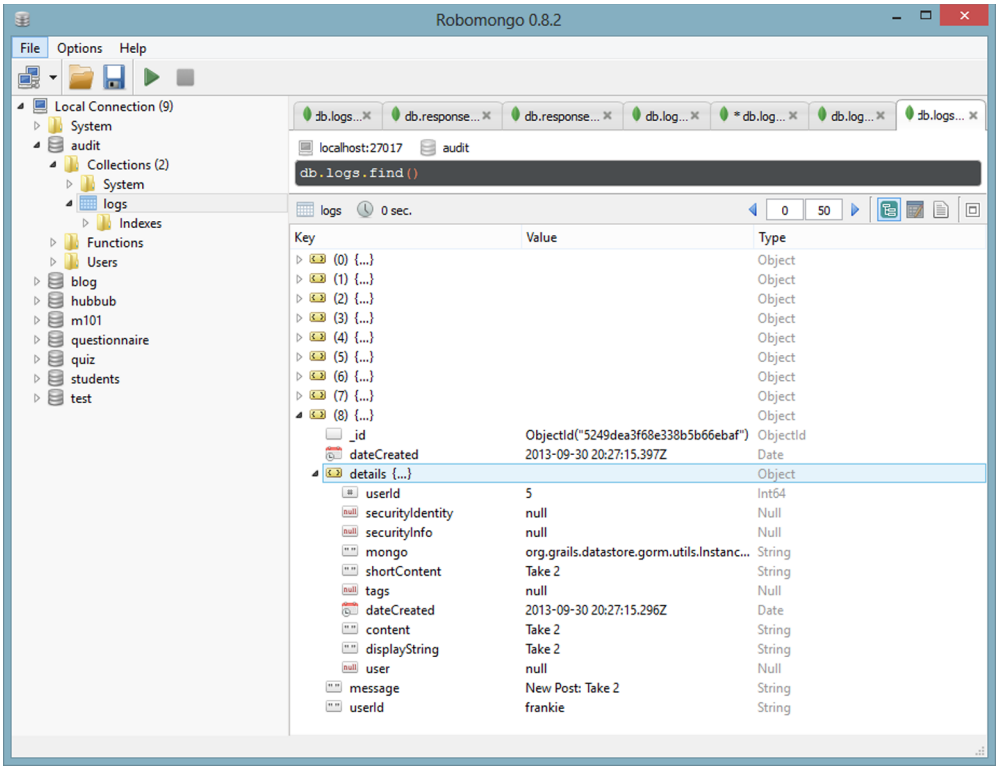


Figure 16.5 Embedded details as a subdocument

In this case, you probably made things too "noisy" with overhead fields. You're better off whitelisting properties (as you feel appropriate) during the assignment to tidy things up. Perhaps something like this:

```
auditEntry.details = newPost.properties['userId',
    'shortContent', 'dateCreated']
```

But what if you want to go beyond embedded maps and embed domain classes? Turns out the MongoDB plugin takes advantage of GORM's standard embedded annotation.

Let's enhance your `AuditEntry` to be taggable. You can tag each audit entry with one or more tags to allow pick up of audit entries that relate to object creation, object access, deletion, and so on. Here's your enhanced `AuditEntry` with the new modeling:

```
class AuditEntry {

    static mapWith = "mongo"

    ObjectId id
    String message
    String userId
    Date dateCreated

    Map details

    static hasMany = [ tags : AuditTag ]

    static embedded = ['tags']
}
```

Uses standard hasMany to say you linked objects

Marks tags as embedded

Now you need to define your simple `AuditTag` and you're ready to persist:

```
package com.grailsinaction

class AuditTag {

    String name

}
```

Once again, after the plumbing is in place, all your normal GORM persistence operations work in the standard relational manner. Here's an enhanced `AuditService` method to add tags to your logging:

```
@grails.events.Listener
def onNewPost(Post newPost){
    def auditEntry = new AuditEntry(message:
        "New Post: ${newPost.shortContent}",
    userId: newPost.user.loginId)
    auditEntry.details = newPost.properties['userId',
        'shortContent', 'dateCreated']
    auditEntry.addToTags(new AuditTag(name: "post"))
    auditEntry.addToTags(new AuditTag(name: "create"))
    auditEntry.addToTags(new AuditTag(name: "user-driven"))
    auditEntry.save(failOnError: true)
}
```

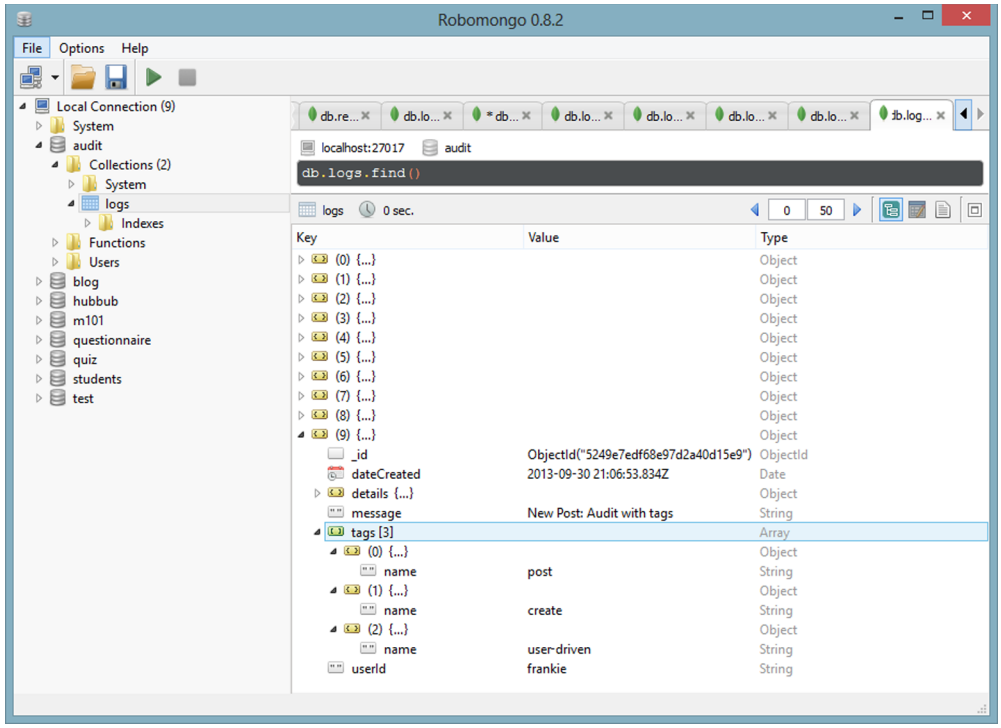


Figure 16.6 Tags nested in subdocuments

Notice you use the standard GORM `addToTags` infrastructure to work with your embedded collection of tags. Once you create a few audits, browse Robomongo to confirm your new tags are nicely nested inside a set of subdocuments, as shown in figure 16.6.

Your three tags seem to be nicely embedded there. You explore how to take advantage of querying those subdocuments in a later section.

How does MongoDB store non-embeddable Grails relationships?

You may wonder what happens when MongoDB/GORM encounters related domain classes (such as `oneToMany`) that aren't marked as embedded. By default, the plugin stores the objects as two separate documents, then uses a MongoDB structure known as a DBRef to provide the link between them.

Remember this has performance implications; you'll now do more than one fetch operation to retrieve the related documents when required (which can happen either lazily or eagerly depending on how you configure your plugin).

Now it's time to turn your attention to one of the most interesting aspects of working with a schemaless database: dynamic attributes.

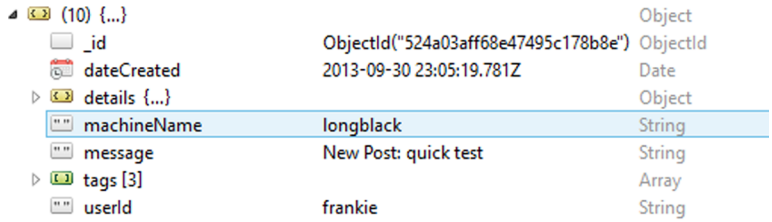


Figure 16.7 Your newly minted audit object now in a MongoDb collection

16.4.7 Dynamic attributes: making up properties as you go along

You've mostly dealt with scenarios where you create a field on a domain class, and then populate it with values. But MongoDB domain classes are happy to have properties dynamically created on them.

To store a `machineName` property on your next `AuditEntry`, you can pretend that the property exists and assign it without having any matching field. This is entirely valid, even without a field definition

```
auditEntry.machineName = InetAddress.localHost.hostName
auditEntry.save(failOnError: true)
```

and your new property is persisted directly to the audit object, as shown in figure 16.7.

You can create your own properties at runtime and add them to your domain object as you go. You could do something like this

```
def dynamicProps = [
    "os-name"       : System.getProperty("os.name"),
    "os-version"    : System.getProperty("os.version"),
    "os-java"       : System.getProperty("java.version")
]
dynamicProps.each { key, value ->
    auditEntry[key] = value
}
auditEntry.save(failOnError: true)
```

which creates the property names dynamically at runtime, as shown in figure 16.8.

You've comprehensively explored all the common dynamic data storage aspects that MongoDB brings to the table. But what about querying all that dynamic data? In the next section you see how MongoDB/GORM makes that painless.

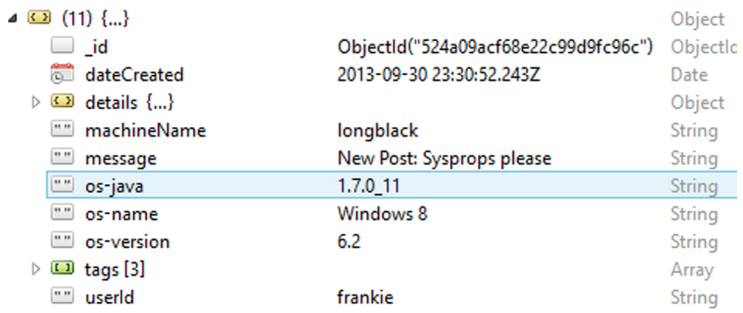


Figure 16.8 New property names created at runtime

16.4.8 Querying MongoDB via standard GORM

This should probably be one of the smallest sections in the book because most of your standard GORM mechanisms apply to MongoDB querying: dynamic finders, criteria queries, named queries, and query by example. You can't use Hibernate's proprietary HQL (or any other Hibernate-specific API), but it's a small price to pay.

The truly amazing thing is that all these query methods work fine with dynamic MongoDB properties. Remember that dynamic `machineName` property you added to `AuditEntry`? You can query on it via normal query APIs

```
def entries = AuditEntry.findByMachineName('longblack')
```

and it returns the full-blown `AuditEntry` objects you expect. You can iterate them in a Grails view to prove how ubiquitous the access is

```
<h1>Recent Audits From Machine: Longblack</h1>
<ul>
    <g:each in="${com.grailsinaction.AuditEntry.
        ➥ findByMachineName('longblack')}" var="auditEntry">
        <li>${auditEntry.message} –
            ${auditEntry.userId} –
            ${auditEntry.dateCreated}
        </li>
    </g:each>
</ul>
```

What if you want to find all the `AuditEntries` that have an embedded tag? Again, all the standard criteria and `where` queries work as you expect. If you want to find all `AuditEntries` with an embedded tag named `post`, you can use a regular `where` query:

```
def entries = AuditEntry.where {
            tags.name == "post"
}.list()
```

and `entries` contains a `List` of `AuditEntry` objects that you can manipulate in whatever way makes sense for your application.

But what if you want to go lower level and do raw MongoDB querying without going through GORM? Even in those scenarios the plugin has you covered. Let's go native.

16.4.9 Working with low-level MongoDB querying

In addition to the GORM standard API, the plugin enhances your domain class with a `collection` property giving you access to the underlying MongoDB collection via the low-level GMongo API. Be warned, though, you're now working with MongoDB objects and not GORM domain classes.

To repeat your previous query using raw MongoDB querying, you can enter

```
def entries = AuditEntry.collection.find(tags: [ name: 'post' ])
```

which returns a list of `DBObjects` (that you can treat as a `Map` if you're reading values). If you need to convert your results back to a domain class, the plugin registers type converters for you, so go ahead and jump in

```
def entries = AuditEntry.collection.find(tags: [ name: 'post' ])
entries.each { entry ->
    AuditEntry auditEntry = entry as AuditEntry
    // and you have yourself a domain class
}
```

If you need to go even lower than the query layer, Grails also injects a GMongo object (https://github.com/poiati/gmongo) on any service and controller classes that define a MongoDB property.

With an injected GMongo instance, you can be as hard-core MongoDB as you like. How about a Mongo Map-Reduce function that counts the number of audit entries per user and then stores that in a new collection called `auditCounts`? The following listing shows an enhancement to your `StatsService` to do that, and then returns a map of `userId` to `auditCount` to boot!

Listing 16.5 Enhancing `StatsService`

```
class StatsService {

    def mongo

    def countAuditMessageByUser() {

        def db = mongo.getDB("audit")
        def result = db.logs.mapReduce("""
    function map() {
        emit(this.userId, this.message)
    }
""",
                """
    function reduce(userId, auditMessages) {
        return auditMessages.length
    }
""",
                "auditCount", [:]
        )

        def countMap = [ : ]

        db.auditCount.find().each { counter ->
            countMap[counter._id] = counter.value
        }

        return countMap
    }

    // rest of StatsService omitted.

}
```

Declares MongoDB handle for injection

Maps function to pair data values

Reduces function to count number of audit entries

Stores result in a collection called auditCount

Transforms auditCount collection to map of userId to count

As you can see, the sky is the limit with an injected MongoDB instance. Here you pass in a `Map` function (in JavaScript because it runs inside MongoDB itself) that maps all the `AuditEntries` in your database as a tuple of `userId` and `message`. You then feed those tuples into your `reduce` function (which is handed a `userId` along with an array of matching messages), and you return a count of those messages back to MongoDB.

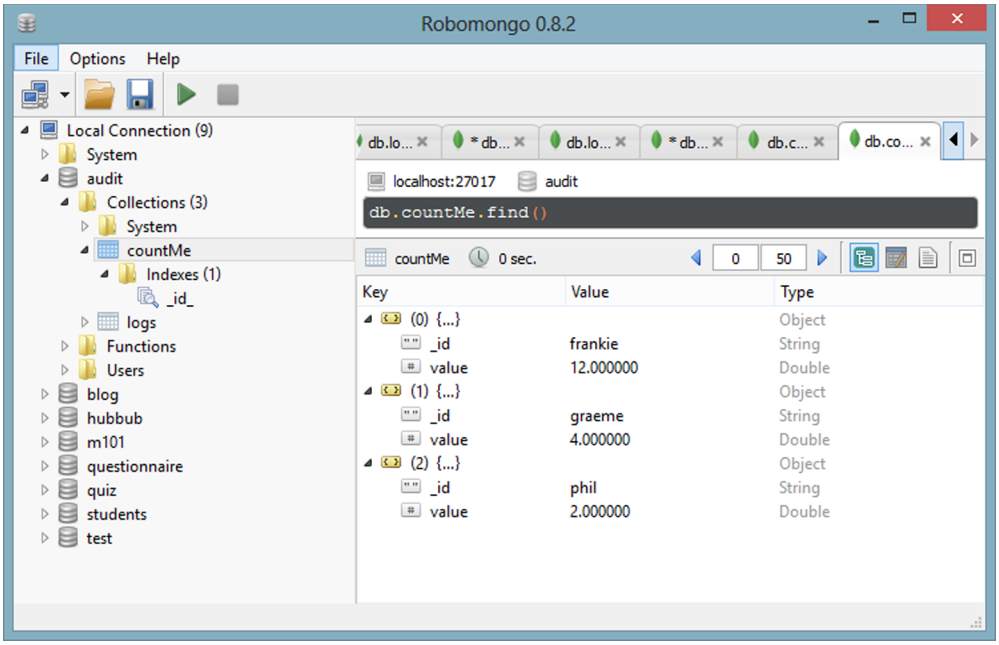


Figure 16.9 The results of our MapReduce operation

After all that Map/Reducing, you're left with a collection of documents that map a `userId` to a count of entries that MongoDB stores in a collection you named `auditCount`. If the collection already exists, MongoDB wipes it out on the next run. You can even browse the results in Robomongo (see figure 16.9).

With insanely hard-core Map/Reduce code under your belt, you're probably deeper into MongoDB than you ever planned to be in an introductory Grails book! Let's spend the last section of the chapter exploring another interesting take on the NoSQL story: graph databases with Neo4j.

16.5 Using Neo4j to work with graph-oriented data

Neo4j is your final stop on your NoSQL explorations, and we chose it because it models data and relationships in a completely different way than anything you've encountered so far. You've seen the relational model, key/value model, and document model, but Neo4j introduces you to modeling data as a graph of connected data.

It's probably been a while since you've played with graph data structures, though you use them behind the scenes every time you use a social networking application such as LinkedIn, Facebook, or Twitter. Let's reintroduce them to you using Hubbub as an example.

When one Hubbub user follows another, you create a link in the database that you model as `firstUser.addToFollowing(targetUser)`. With each user following several

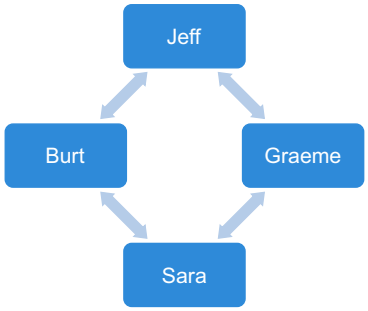


Figure 16.10 Social networks contain graphs of data where items are linked by relationships.

other users in the system, you soon end up with a web or "graph" of relationships such as the one shown in figure 16.10.

Graph databases such as Neo4j specialize in modeling these types of relationships and provide high performance tools for querying them. Need to find out which users are within three degrees of separation from a particular user? That's expensive to do in a relational data source but bread and butter for Neo4j. Let's install it and whip up a social graph visualization for Hubbub.

16.5.1 Installing and configuring the Neo4j plugin

Your first step in getting Hubbub into graph database territory is to install the Grails Neo4j plugin (http://grails.org/plugin/neo4j). A quick update of your /grails-app/conf/BuildConfig.groovy should sort that out. At the time of writing, the current version is 1.0.1, so add it to your list of plugins:

```
plugins {
    ...
    compile ":neo4j:1.0.1"
    ...
}
```

By default the Neo4j plugin sparks up an embedded version of Neo4j server that runs in the same JVM as your Grails app. That's perfect for your experimentation, but you can always override the defaults by customizing your /grails-app/conf/Config.groovy. The default place in which the plugin stores your Neo4j database is /var/neo4j, so if you're on a local Windows box, tweak that to something that makes more sense:

```
grails {
    neo4j {
        type = "embedded"
        location = "/data/neo4j"
    }
}
```

With the plugin installed and optionally configured, let's get under way teaching you Neo4j parlance as you implement Hubbub's social graph searcher.

16.5.2 Neo4j domain classes: combining with Hibernate

You can use Neo4j as your primary data source, if you like. In that case, as in MongoDB, it's a matter of removing your Hibernate plugin from /grails-app/conf/BuildConfig.groovy and you're ready to run. No special domain class markup is required.

In this case, let's supplement your existing Hibernate (and MongoDB) domain classes with a few Neo4j-specific domain classes. The way you mark a domain class for persistence in Neo4j is similar to what you did with MongoDB—add a custom `mapsWith` property.

Let's create a domain class that you can use to keep the social networking graph for Hubbub. Let's start with the minimal set of data you might keep to store a graph—`userId`s and their relationships:

```
package com.grailsinaction

class UserGraph {

    static mapWith = "neo4j"

    String loginId

    static hasMany = [ following : User ]

    static constraints = {
        loginId blank: false
    }
}
```

Now that you have your domain class in place, you need code to populate it with real data so you have something to query. Let's write the glue to perform the synchronization.

16.5.3 Populating Hubbub's social graph

You need a way to sync your domain class with your existing user relationships. Let's create a `GraphController` to house all your graph interactions, and perhaps put in a scrappy little `sync()` method to convert your list of users and their followers into a graph of `UserGraph` objects, as shown in the following listing.

Listing 16.6 Creating a `GraphController` and `sync` method

```
package com.grailsinaction

class GraphController {

    private UserGraph getOrCreateMatchingUserGraph(User user) {

        UserGraph matchingGraphUser = UserGraph.findByLoginId(user.loginId)
        if (!matchingGraphUser) {
            matchingGraphUser = new UserGraph(loginId: user.loginId)
            matchingGraphUser.save(failOnError: true)
            if (user.profile?.fullName) {
                matchingGraphUser.fullName = user.profile.fullName
            }
        }
        return matchingGraphUser
    }
```

```
    def sync() {

        log.debug("Starting sync process...")

        int syncCount = 0
        int linkCount = 0

        UserGraph.list()*.delete() // go nuclear
        User.list().each { user ->

            UserGraph matchingGraphUser =
                getOrCreateMatchingUserGraph(user)
            user.following.each { nextFollowing ->
                UserGraph matchingFollow =
     getOrCreateMatchingUserGraph(nextFollowing)
                matchingGraphUser.addToFollowing(matchingFollow)
                linkCount++
            }
            syncCount++
        }
        render text: "<html>Sync complete. Synced ${syncCount} users with
     ${linkCount} links at ${new Date()}</html>",
            contentType: "text/html"

    }
}
```

Nothing particularly exciting is going on there. You use standard GORM domain logic with the odd `addToFollowing()` and `save()` calls. Under the covers the Neo4j plugin persists all those objects for you.

You may notice that you use dynamic properties (as you did in MongoDB) to store the user's full name on the `UserGraph` node because that comes in handy for rendering later on. One gotcha with the current version of the plugin is that dynamic properties can be set only after `save()` is called, hence the unusual placement in your source code.

It's time to experiment with walking the tree and rendering nodes.

16.5.4 Walking and visualizing the graph with Cypher

Neo4j offers two ways to query the graph in object style:

- Its own SQL-like query language called Cypher
- A code-centric mechanism exposed via a traversal API

All Neo4j domain classes are enhanced with several variations of the `cypher()` and `traverse()` methods to make things easy.

You start your journey using the Cypher query language to find friends of friends of a user. The idea is that you can pass in a user, find all their friends, then find all the friends that are friends with them. In a relational world, you'd need many joins, but as you'll see, Neo4j makes that a one-liner.

Let's implement the `friendsOfFriends()` action in your graph controller, as shown in the following listing, then we'll show you how it all hangs together.

Listing 16.7 Implementing `friendsOfFriends()` in your graph controller

```
def friendsOfFriends() {

        if (params.id) {
            UserGraph startingUser = UserGraph.findByLoginId(params.id)
            if (startingUser) {
                def resultsTable = startingUser.cypher(
                    "start myself=node({this})
                     MATCH myself-[:following]->friend-[:following]->fof
                     WHERE fof.loginId <> myself.loginId
                     RETURN myself, friend, fof")
                [resultsTable: resultsTable]
            } else {
                response.sendError(404)
            }
        } else {
            response.sendError(404)
        }

    }
```

Finds matching UserGraph using GORM dynamic finder ❶

Runs Cypher query to find followers of followers ❷

Passes results to view ❸

This is familiar territory. You grab the user's ID off the incoming URL and attempt to find a matching `UserGraph` in your Neo4j database using standard GORM dynamic finders ❶. If you can locate them, you invoke the Neo4j `cypher` method ❷ on that domain class instance, passing it complex-looking Cypher code (which we'll get to in a moment). Finally, you pass any results, which are returned as a table-like structure, through to the view for rendering ❸. We'll look at that rendering code in a minute, but for now, let's break down that Cypher query so it becomes less magical.

First, let's reformat the query so you can see the individual clauses:

```
start myself=node({this})
MATCH myself-[:following]->friend-[:following]->fof
WHERE fof.loginId <> myself.loginId
RETURN myself, friend, fof
```

If you restate each clause in plainer English, this query says, "Start at the current node, which I'm going to now alias as 'myself'. Then match all the nodes that have a 'following' relationship with me, and alias them as a 'friend'. Then match all the nodes that have a 'following' relationship with 'friend', and alias them as 'fof' (friend of friend). Make sure that my fof.loginId doesn't match my own loginId because I don't want to display cases where my friends follow me back. Finally, return a table with three columns: myself, friend, and fof."

Phew! It's a mouthful of a query, but let's make it clearer by putting it to use in a view. To render a table that outputs you, your friend, and their friends, you iterate that `resultsTable`. The following listing shows what you may find in a friendsOf-Friends.gsp.

Listing 16.8 Creating a view for the `resultsTable`

```
<h1>Friends Of Friends</h1>
    <table>
        <tr>
            <th>User</th><th>Is A Friend Of</th><th>Who Is A Friend Of</th>
        </tr>
        <g:each in="${resultsTable}" var="row">
             <tr>

<td><g:link action="friendsOfFriends"
    id="${row.myself.loginId}">${row.myself.fullName}
</g:link></td>
<td><g:link action="friendsOfFriends"
     id="${row.friend.loginId}">${row.friend.fullName}
</g:link></td>
<td><g:link action="friendsOfFriends"
    id="${row.fof.loginId}">${row.fof.fullName}
</g:link></td>
             </tr>
        </g:each>
    </table>
```

Each of the rows in that `resultsTable` exposes you, your friend, and friend-of-friend objects whose properties you can inspect to get back your underlying attributes. What you iterate here are the underlying node objects. If you want to get back to the matching domain objects (for example, for manipulation), you can take advantage of another domain class convenience method and call `UserGraph.createInstanceForNode(row.myself)`, which gives you back the domain class instance matching this node.

You create links on each of those users so you can keep exploring who's linked to whom. You also used that dynamic `fullName` property that you previously squirreled away on each node. Figure 16.11 shows the view in action for `loginId` jeff (http://localhost:8080/hubbub/graph/friendsOfFriends/jeff):

That's an impressive way to browse relationships and only scratches the surface of what Cypher can do. If you want to go deeper, fantastic docs (with working examples) are on the Neo4j site (http://docs.neo4j.org/chunked/milestone/cypher-introduction.html).

Friends Of Friends

User	Is A Friend Of	Who Is A Friend Of
Jeff Brown	Burt Beckwith	Sara Miles
Jeff Brown	Burt Beckwith	Graeme Rocher
Jeff Brown	Graeme Rocher	Burt Beckwith
Jeff Brown	Graeme Rocher	Dillon Jessop

Figure 16.11 Viewing the friends of Jeff

Querying friends of friends is impressive, but what if you want to walk the entire object graph displaying every relationship? For that use case it may be time to drop down to the Neo4j traversal API to see what's achievable in code.

16.5.5 Walking the entire graph

You've experimented with the Cypher query language through the domain class instance `cypher()` method, and now let's look at the API equivalent by seeing what's possible through the `traverse()` method.

Let's implement a `walk()` action on your graph controller that starts at a given node, and then traverses its `following` relationships until it runs out of nodes. Depending on your starting node, and who's following whom, you may even see the entire system!

The following listing shows what your `walk()` action looks like.

Listing 16.9 Using `walk()` to find relationships

```
import org.grails.datastore.gorm.neo4j.GrailsRelationshipTypes
import org.neo4j.graphdb.*

def walk() {

        if (params.id) {
            UserGraph startingUser = UserGraph.findByLoginId(params.id)
            if (startingUser) {
                def followingRel = startingUser.node.relationships.
                    ➥ find { it.type.name == 'following' }
                def nodeList =
    startingUser.traverse(Traverser.Order.BREADTH_FIRST,
                        StopEvaluator.END_OF_GRAPH,
                        ReturnableEvaluator.ALL,
                        followingRel.type, Direction.OUTGOING)
                [nodeList: nodeList]
            } else {
                response.sendError(404)
            }
        } else {
            response.sendError(404)
        }

    }
```

The `nodeList` returns a list of all the `UserGraph` nodes that Neo4j found by traversing outward links. But it's no fun if you can't see them, so let's add a walk.gsp view so you can see exactly what's happening, as shown in the following listing.

Listing 16.10 Viewing the list of users

```
<h1>Walking The Graph</h1>
  <table>
      <tr>
          <th>User</th><th>Following</th>
      </tr>
```

```
        <g:each in="${nodeList}" var="node">
             <tr>
                 <td><g:link action="walk"
                      id="${node.loginId}">${node.fullName}</g:link>
                 </td>
                 <td>
                     <ul>
                     <g:each in="${node.following}" var="following">
                         <li>
                             <g:link action="walk"
            id="${following.loginId}">${following.fullName}</g:link>
                         </li>
                     </g:each>
                     </ul>
             </tr>
        </g:each>
    </table>
```

In this view you display the node you found, and all the nodes that node follows. Figure 16.12 shows the output for user jeff.

If you notice carefully, the table can be read from the top down. Jeff follows Burt and Graeme, so they're the next two nodes you see in the table. Then rinse and repeat all the way down.

In this graph, Dillon is followed by Graeme, but Dillon himself doesn't follow anyone. If you click the Dillon link, you shouldn't see any outgoing links in your traversal. In fact, figure 16.13 shows exactly what that output looks like.

Using the traversal API offers you powerful features. You traversed OUTGOING relationships, but you can traverse INCOMING, OUTGOING, or BOTH depending on how you want to navigate your tree.

Walking The Graph

User	Following
Jeff Brown	• Burt Beckwith • Graeme Rocher
Graeme Rocher	• Burt Beckwith • Dillon Jessop • Jeff Brown
Burt Beckwith	• Sara Miles • Graeme Rocher
Dillon Jessop	
Sara Miles	• Burt Beckwith • Frankie Goes to Hollywood
Frankie Goes to Hollywood	• Phil Potts
Phil Potts	• Frankie Goes to Hollywood • Sara Miles

Figure 16.12 Viewing all the friends linked to Jeff

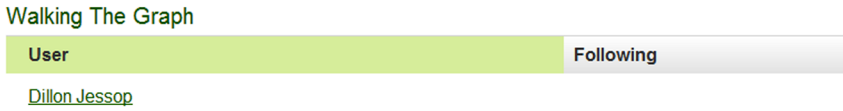


Figure 16.13 Viewing a friend with no outgoing links

With a good sense of what's achievable via Neo4j's low-level API, it's time to wrap up your tour of popular NoSQL technologies in Grails.

16.6 Summary and best practices

We covered NoSQL territory in this chapter, introducing three of the most dominant NoSQL technologies available in the space today:

- *Redis*—for persistent key/value storage
- *MongoDB*—for document-oriented storage
- *Neo4j*—for graph-based storage and traversal

No doubt your head is spinning! This chapter was designed to give you a basic level of exposure to all three types of stores, so you can decide which ones you may like to explore further. Before we leave back-end territory and move on to testing and compiling in chapter 17, let's review a few key best practices from this chapter:

- *Redis is a data-structure server.* While you can use Redis as a persistent hash table, it shines when you take advantage of its high-performance data structures, such as lists, hashes, and sorted sets.
- *Use Redis to back Grails caching.* Now that caching services are built into the Grails platform, don't forget you can easily back your caches with Redis via the Grails Redis Cache plugin.
- *Experiment with MongoDB native queries via query tools.* Using a GUI tool such as Robomongo gives you freedom to experiment with Mongo queries or browse the results of previous Grails database operations. Don't be afraid to use these GUI tools to learn more about optimizing your Mongo queries. This approach can save you time.
- *Always provide an ObjectId field on your MongoDB domain classes.* Remember that if you don't provide your own `ObjectId` field, Grails supplies a long-based one. This can hamper your clustering options later, so bite the bullet and put an `ObjectId` ID field on all your domain classes from the get-go.
- *You can always fall back to GMongo.* The Grails Mongo integration is complete, but if you ever hit an edge case not supported out of the box, don't forget that you can always drop back to straight GMongo code and do anything you need to.
- *Consider a graph database.* If your application works with graph-based data structures (such as your social networking app Hubbub), consider storing your data

the way it wants to be stored. You end up with less code to maintain, and you won't have to worry about endless tuning of relational databases.

- *Learn Neo4j's Cypher query language.* Using Cypher to query a graph gives you a fast and self-describing mechanism for rich graph queries. Take the time to work through the Neo4j documentation and learn the basics of the language. The docs are great, and in-browser tools in the documentation let you experiment. It's worth the investment.

In the next chapter, we explore the processes involved in compiling, testing, and running your app.

JAVA

Grails IN ACTION Second Edition

Smith • Ledbrook

It may be time for you to stop reconfiguring, rewriting, and recompiling your Java web apps. Grails, a Groovy-powered web framework, hides all that busy work so you can concentrate on what your applications do, not how they're built. In addition to its famously intuitive dev environment and seamless integration with Spring and Hibernate, the new Grails 2.3 adds improved REST support, better protection against attacks from the web, and better dependency resolution.

Grails in Action, Second Edition is a comprehensive introduction to Grails 2. In this totally revised edition you'll master Grails as you apply TDD techniques to a full-scale example (a Twitter clone). Along the way you'll learn "single-page web app" techniques, work with NoSQL back ends, integrate with enterprise messaging, and implement a RESTful API.

What's Inside

- Covers Grails 2.3 from the ground up
- Agile delivery and testing using Spock
- How to use and manage plugins
- Tips and tricks from the trenches

No Java or Groovy knowledge is required. Some web development and OOP experience is helpful.

There's no substitute for experience: **Glen Smith** and **Peter Ledbrook** have been fixtures in the Grails community, contributing code, blogging, and speaking at conferences worldwide, since Grails 0.2.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GrailsinActionSecondEdition

"Provides a solid foundation ... covers best practices and background knowledge."
—From the Foreword by Dierk König, author of *Groovy in Action*

"The canonical guide to Grails."
—Jerry Gaines, 4impact Group

"Packed with practical examples."
—Pratap Chatterjee
Karolinska Institute

"The best resource to help you get ridiculously productive!"
—Michael A. Angelo
Laird Technologies

ISBN 13: 978-1-617290-96-1
ISBN 10: 1-617290-96-3
54999
9 781617 290961

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]