



Learn by doing: less theory, more results

Ruby and MongoDB Web Development

Create dynamic web applications by combining the power
of Ruby and MongoDB

Beginner's Guide

Gautam Rege

www.cxy808.com

[PACKT] open source 
PUBLISHING community experience distilled

Ruby and MongoDB Web Development Beginner's Guide

Create dynamic web applications by combining
the power of Ruby and MongoDB

Gautam Rege



BIRMINGHAM - MUMBAI

Ruby and MongoDB Web Development Beginner's Guide

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2012

Production Reference: 1180712

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-502-3

www.packtpub.com

Cover Image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Author

Gautam Rege

Project Coordinator

Leena Purkait

Reviewers

Bob Chesley

Ayan Dave

Michael Kohl

Srikanth AD

Proofreader

Linda Morris

Indexer

Hemangini Bari

Acquisition Editor

Kartikey Pandey

Graphics

Valentina D'silva

Manu Joseph

Lead Technical Editor

Dayan Hyames

Production Coordinator

Prachali Bhiwandkar

Technical Editor

Prashant Salvi

Cover Work

Prachali Bhiwandkar

Copy Editors

Alfida Paiva

Laxmi Subramanian

About the Author

Gautam Rege has over twelve years of experience in software development. He is a Computer Engineer from Pune Institute of Computer Technology, Pune, India. After graduating in 2000, he worked in various Indian software development companies until 2002, after which, he settled down in Veritas Software (now Symantec). After five years there, his urge to start his own company got the better of him and he started Josh Software Private Limited along with his long time friend Sethupathi Asokan, who was also in Veritas.

He is currently the Managing Director at Josh Software Private Limited. Josh in Hindi (his mother tongue) means "enthusiasm" or "passion" and these are the qualities that the company culture is built on. Josh Software Private Limited works exclusively in Ruby and Ruby related technologies, such as Rails – a decision Gautam and Sethu (as he is lovingly called) took in 2007 and it has paid rich dividends today!

Acknowledgement

I would like to thank Sethu, my co-founder at Josh, for ensuring that my focus was on the book, even during the hectic activities at work. Thanks to Satish Talim, who encouraged me to write this book and Sameer Tilak, for providing me with valuable feedback while writing this book! Big thanks to Michael Kohl, who was of great help in ensuring that every tiny technical detail was accurate and rich in content. I have become "technically mature" because of him!

The book would not have been completed without the positive and unconditional support from my wife, Vaibhavi and daughter, Swara, who tolerated a lot of busy weekends and late nights where I was toiling away on the book. Thank you so much!

Last, but not the least, a big thank you to Kartikey, Leena, Dayan, Ayan, Prashant, and Vrinda from Packt, who ensured that everything I did was in order and up to the mark.

About the Reviewers

Bob Chesley is a web and database developer of around twenty years currently concentrating on JavaScript cross platform mobile applications and SaaS backend applications that they connect to. Bob is also a small boat builder and sailor, enjoying the green waters of the Tampa Bay area. He can be contacted via his web site (www.nhsoftwerks.com) or via his blog (www.cfmeta.com) or by email at bob.chesley@nhsoftwerks.com.

Ayan Dave is a software engineer with eight years of experience in building and delivering high quality applications using languages and components in JVM ecosystem. He is passionate about software development and enjoys exploring open source projects. He is enthusiastic about Agile and Extreme Programming and frequently advocates for them. Over the years he has provided consulting services to several organizations and has played many different roles. Most recently he was the "Architectus Oryzus" for a small project team with big ideas and subscribes to the idea that running code is the system of truth.

Ayan has a Master's degree in Computer Engineering from the University of Houston - Clear Lake and holds PMP, PSM-1 and OCMJEA certifications. He is also a speaker on various technical topics at local user groups and community events. He currently lives in Columbus, Ohio and works with Quick Solutions Inc. In the digital world he can be found at <http://daveayan.com>.

Michael Kohl got interested in programming, and the wider IT world, at the young age of 12. Since then, he worked as a systems administrator, systems engineer, Linux consultant, and software developer, before crossing over into the domain of IT security where he currently works. He's a programming language enthusiast who's especially enamored with functional programming languages, but also has a long-standing love affair with Ruby that started around 2003. You can find his musings online at <http://citizen428.net>.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ◆ Fully searchable across every book published by Packt
- ◆ Copy and paste, print and bookmark content
- ◆ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Installing MongoDB and Ruby	11
Installing Ruby	12
Using RVM on Linux or Mac OS	12
The RVM games	16
The Windows saga	17
Using rbenv for installing Ruby	17
Installing MongoDB	18
Configuring the MongoDB server	19
Starting MongoDB	19
Stopping MongoDB	21
The MongoDB CLI	21
Understanding JavaScript Object Notation (JSON)	21
Connecting to MongoDB using Mongo	22
Saving information	22
Retrieving information	23
Deleting information	24
Exporting information using mongoexport	24
Importing data using mongoimport	25
Managing backup and restore using mongodump and mongorestore	25
Saving large files using mongofiles	26
bsondump	28
Installing Rails/Sinatra	28
Summary	29
Chapter 2: Diving Deep into MongoDB	31
Creating documents	32
Time for action – creating our first document	32
NoSQL scores over SQL databases	33
Using MongoDB embedded documents	34

Table of Contents

Time for action – embedding reviews and votes	35
Fetching embedded objects	36
Using MongoDB document relationships	36
Time for action – creating document relations	37
Comparing MongoDB versus SQL syntax	38
Using Map/Reduce instead of join	40
Understanding functional programming	40
Building the map function	40
Time for action – writing the map function for calculating vote statistics	41
Building the reduce function	41
Time for action – writing the reduce function to process emitted information	42
Understanding the Ruby perspective	43
Setting up Rails and MongoDB	43
Time for action – creating the project	43
Understanding the Rails basics	44
Using Bundler	44
Why do we need the Bundler	44
Setting up Sodibee	45
Time for action – start your engines	45
Setting up Mongoid	46
Time for action – configuring Mongoid	47
Building the models	48
Time for action – planning the object schema	48
Testing from the Rails console	52
Time for action – putting it all together	52
Understanding many-to-many relationships in MongoDB	56
Using embedded documents	57
Time for action – adding reviews to books	57
Choosing whether to embed or not to embed	58
Time for action – embedding Lease and Purchase models	59
Working with Map/Reduce	60
Time for action – writing the map function to calculate ratings	63
Time for action – writing the reduce function to process the emitted results	64
Using Map/Reduce together	64
Time for action – working with Map/Reduce using Ruby	65
Summary	68
Chapter 3: MongoDB Internals	69
Understanding Binary JSON	70
Fetching and traversing data	71
Manipulating data	71

What is ObjectId?	71
Documents and collections	71
Capped collections	72
Dates in MongoDB	72
JavaScript and MongoDB	72
Time for action – writing our own custom functions in MongoDB	73
Ensuring write consistency or "read your writes"	73
How does MongoDB use its memory-mapped storage engine?	74
Advantages of write-ahead journaling	74
Global write lock	74
Transactional support in MongoDB	75
Understanding embedded documents and atomic updates	75
Implementing optimistic locking in MongoDB	75
Time for action – implementing optimistic locking	76
Choosing between ACID transactions and MongoDB transactions	77
Why are there no joins in MongoDB?	77
Summary	79
Chapter 4: Working Out Your Way with Queries	81
Searching by fields in a document	81
Time for action – searching by a string value	82
Querying for specific fields	84
Time for action – fetching only for specific fields	84
Using skip and limit	86
Time for action – skipping documents and limiting our search results	86
Writing conditional queries	87
Using the \$or operator	88
Time for action – finding books by name or publisher	88
Writing threshold queries with \$gt, \$lt, \$ne, \$lte, and \$gte	88
Time for action – finding the highly ranked books	89
Checking presence using \$exists	89
Searching inside arrays	90
Time for action – searching inside reviews	90
Searching inside arrays using \$in and \$nin	91
Searching for exact matches using \$all	92
Searching inside hashes	92
Searching inside embedded documents	93
Searching with regular expressions	93
Time for action – using regular expression searches	94
Summary	97

Table of Contents

Chapter 5: Ruby DataMappers: Ruby and MongoDB Go Hand in Hand	99
Why do we need Ruby DataMappers	99
The mongo-ruby-driver	100
Time for action – using mongo gem	101
The Ruby DataMappers for MongoDB	103
MongoMapper	104
Mongoid	104
Setting up DataMappers	104
Configuring MongoMapper	104
Time for action – configuring MongoMapper	105
Configuring Mongoid	107
Time for action – setting up Mongoid	107
Creating, updating, and destroying documents	110
Defining fields using MongoMapper	110
Defining fields using Mongoid	111
Creating objects	111
Time for action – creating and updating objects	111
Using finder methods	112
Using find method	112
Using the first and last methods	113
Using the all method	113
Using MongoDB criteria	113
Executing conditional queries using where	113
Time for action – fetching using the where criterion	114
Revisiting limit, skip, and offset	115
Understanding model relationships	116
The one to many relation	116
Time for action – relating models	116
Using MongoMapper	116
Using Mongoid	117
The many-to-many relation	118
Time for action – categorizing books	118
MongoMapper	118
Mongoid	119
Accessing many-to-many with MongoMapper	120
Accessing many-to-many relations using Mongoid	120
The one-to-one relation	121
Using MongoMapper	122
Using Mongoid	122
Time for action – adding book details	123
Understanding polymorphic relations	124
Implementing polymorphic relations the wrong way	124
Implementing polymorphic relations the correct way	124

Table of Contents

Time for action – managing the driver entities	125
Time for action – creating vehicles using basic polymorphism	129
Choosing SCI or basic polymorphism	132
Using embedded objects	133
Time for action – creating embedded objects	134
Using MongoMapper	134
Using Mongoid	134
Using MongoMapper	137
Using Mongoid	137
Reverse embedded relations in Mongoid	137
Time for action – using embeds_one without specifying embedded_in	138
Time for action – using embeds_many without specifying embedded_in	139
Understanding embedded polymorphism	140
Single Collection Inheritance	141
Time for action – adding licenses to drivers	141
Basic embedded polymorphism	142
Time for action – insuring drivers	142
Choosing whether to embed or to associate documents	144
Mongoid or MongoMapper – the verdict	145
Summary	146
Chapter 6: Modeling Ruby with Mongoid	147
Developing a web application with Mongoid	147
Setting up Rails	148
Time for action – setting up a Rails project	148
Setting up Sinatra	149
Time for action – using Sinatra professionally	151
Understanding Rack	156
Defining attributes in models	157
Accessing attributes	158
Indexing attributes	158
Unique indexes	159
Background indexing	159
Geospatial indexing	159
Sparse indexing	160
Dynamic fields	160
Time for action – adding dynamic fields	160
Localization	162
Time for action – localizing fields	162
Using arrays and hashes in models	164
Embedded objects	165

Table of Contents

Defining relations in models	165
Common options for all relations	165
:klass option	166
:inverse_of option	166
:name option	166
Relation-specific options	166
Options for has_one	167
:as option	167
:autosave option	168
:dependent option	168
:foreign_key option	168
Options for has_many	168
:order option	168
Options for belongs_to	169
:index option	169
:polymorphic option	169
Options for has_and_belongs_to_many	169
:inverse_of option	170
Time for action – configuring the many-to-many relation	171
Time for action – setting up the following and followers relationship	172
Options for :embeds_one	175
:cascade_callbacks option	175
:cyclic	175
Time for action – setting up cyclic relations	175
Options for embeds_many	176
:versioned option	176
Options for embedded_in	176
:klass option	177
Managing changes in models	178
Time for action – changing models	178
Mixing in Mongoid modules	179
The Paranoia module	180
Time for action – getting paranoid	180
Versioning	182
Time for action – including a version	182
Summary	185
Chapter 7: Achieving High Performance on Your Ruby Application with MongoDB	187
Profiling MongoDB	188
Time for action – enabling profiling for MongoDB	188
Using the explain function	190
Time for action – explaining a query	190
Using covered indexes	193

Table of Contents

Time for action – using covered indexes	193
Other MongoDB performance tuning techniques	196
Using mongostat	197
Understanding web application performance	197
Web server response time	197
Throughput	198
Load the server using httpperf	198
Monitoring server performance	199
End-user response and latency	202
Optimizing our code for performance	202
Indexing fields	202
Optimizing data selection	203
Optimizing and tuning the web application stack	203
Performance of the memory-mapped storage engine	203
Choosing the Ruby application server	204
Passenger	204
Mongrel and Thin	204
Unicorn	204
Increasing performance of Mongoid using bson_ext gem	204
Caching objects	205
Memcache	205
Redis server	205
Summary	206
Chapter 8: Rack, Sinatra, Rails, and MongoDB – Making Use of them All	207
Revisiting Sodibee	208
The Rails way	208
Setting up the project	208
Modeling Sodibee	210
Time for action – modeling the Author class	210
Time for action – writing the Book, Category and Address models	211
Time for action – modeling the Order class	212
Understanding Rails routes	213
What is the RESTful interface?	214
Time for action – configuring routes	214
Understanding the Rails architecture	215
Processing a Rails request	216
Coding the Controllers and the Views	217
Time for action – writing the AuthorsController	218
Solving the N+1 query problem using the includes method	219
Relating models without persisting them	220
Designing the web application layout	223

Table of Contents

Time for action – designing the layout	223
Understanding the Rails asset pipeline	230
Designing the Authors listing page	231
Time for action – listing authors	231
Adding new authors and their books	234
Time for action – adding new authors and books	234
The Sinatra way	240
Time for action – setting up Sinatra and Rack	240
Testing and automation using RSpec	243
Understanding RSpec	244
Time for action – installing RSpec	244
Time for action – sporking it	246
Documenting code using YARD	247
Summary	250
Chapter 9: Going Everywhere – Geospatial Indexing with MongoDB	251
What is geolocation	252
How accurate is a geolocation	253
Converting geolocation to geocoded coordinates	253
Identifying the exact geolocation	254
Storing coordinates in MongoDB	255
Time for action – geocoding the Address model	255
Testing geolocation storage	257
Time for action – saving geolocation coordinates	257
Using geocoder to update coordinates	258
Time for action – using geocoder for storing coordinates	258
Firing geolocation queries	260
Time for action – finding nearby addresses	260
Using mongoid_spacial	262
Time for action – firing near queries in Mongoid	262
Differences between \$near and \$geoNear	263
Summary	264
Chapter 10: Scaling MongoDB	265
High availability and failover via replication	266
Implementing the master/slave replication	266
Time for action – setting up the master/slave replication	266
Using replica sets	271
Time for action – implementing replica sets	272
Recovering from crashes – failover	277
Adding members to the replica set	277
Implementing replica sets for Sodibee	278

Table of Contents

Time for action – configuring replica sets for Sodibee	278
Implementing sharding	283
Creating the shards	284
Time for action – setting up the shards	284
Configuring the shards with a config server	285
Time for action – starting the config server	285
Setting up the routing service – mongos	286
Time for action – setting up mongos	286
Testing shared replication	288
Implementing Map/Reduce	289
Time for action – planning the Map/Reduce functionality	290
Time for action – Map/Reduce via the mongo console	291
Time for action – Map/Reduce via Ruby	293
Performance benchmarking	295
Time for action – iterating Ruby objects	295
Summary	298
Pop Quiz Answers	299
Index	301

Preface

And then there was light – a lightweight database! How often have we all wanted some database that was "just a data store"? Sure, you can use it in many complex ways but in the end, it's just a plain simple data store. Welcome MongoDB!

And then there was light – a lightweight language that was fun to program in. It supports all the constructs of a pure object-oriented language and is fun to program in. Welcome Ruby!

Both MongoDB and Ruby are the fruits of people who wanted to simplify things in a complex world. Ruby, written by Yukihiro Matsumoto was made, picking the best constructs from Perl, SmallTalk and Scheme. They say Matz (as he is called lovingly) "writes in C so that you don't have to". Ruby is an object-oriented programming language that can be summarized in one word: fun!



It's interesting to know that Ruby was created as an "object-oriented scripting language". However, today Ruby can be compiled using JRuby or Rubinius, so we could call it a programming language.

MongoDB has its roots from the word "humongous" and has the primary goal to manage humongous data! As a NoSQL database, it relies heavily on data stored as key-value pairs.

Wait! Did we hear NoSQL – (also pronounced as No Sequel or No S-Q-L)? Yes! The roots of MongoDB lie in its data not having a structured format! Even before we dive into Ruby and MongoDB, it makes sense to understand some of these basic premises:

- ◆ NoSQL
- ◆ Brewer's CAP theorem
- ◆ **Basically Available, Soft-state, Eventually-consistent (BASE)**
- ◆ ACID or BASE

Understanding NoSQL

When the world was living in an age of SQL gurus and Database Administrators with expertise in stored procedures and triggers, a few brave men dared to rebel. The reason was "simplicity". SQL was good to use when there was a structure and a fixed set of rules. The common databases such as Oracle, SQL Server, MySQL, DB2, and PostgreSQL, all promoted SQL – referential integrity, consistency, and atomic transactions. One of the SQL based rebels - SQLite decided to be really "lite" and either ignored most of these constructs or did not enforce them based on the premise: "Know what you are doing or beware".

Similarly, NoSQL is all about using simple keys to store data. Searching keys uses various hashing algorithms, but at the end of the day all we have is a simple data store!

With the advent of web applications and crowd sourcing web portals, the mantra was "more scalable than highly available" and "more speed instead of consistency". Some web applications may be okay with these and others may not. What is important is that there is now a choice and developers can choose wisely!

It's interesting to note that "key-value pair" databases have existed from the early 80's – the earliest to my knowledge being Berkeley DB – blazingly fast, light-weight, and a very simple library to use.

Brewer's CAP theorem

Brewer's CAP theorem states that any distributed computer system can support only any two among consistency, atomicity, and partition tolerance.

- ◆ **Consistency** deals with consistency of data or referential integrity
- ◆ **Atomicity** deals with transactions or a set of commands that execute as "all or nothing"
- ◆ **Partition tolerance** deals with distributed data, scaling and replication

There is sufficient belief that any database can guarantee any two of the above. However, the essence of the CAP theorem is not to find a solution to have all three behaviors, but to allow us to look at designing databases differently based on the application we want to build!

For example, if you are building a **Core Banking System (CBS)**, consistency and atomicity are extremely important. The CBS must guarantee these two at the cost of partition tolerance. Of course, a CBS has its failover systems, backup, and live replication to guarantee zero downtime, but at the cost of additional infrastructure and usually a single large instance of the database.

A heavily accessed information web portal with a large amount of data requires speed and scale, not consistency. Does the order of comments submitted at the same time really matter? What matters is how quickly and consistently the data was delivered. This is a clear case of consistency and partition tolerance at the cost of atomicity.



An excellent article on the CAP theorem is at
[http://www.julianbrowne.com/article/viewer/
brewers-cap-theorem](http://www.julianbrowne.com/article/viewer/brewers-cap-theorem).



What are BASE databases?

"Basically Available, Soft-state, Eventually-consistent"!!

Just the name suggests, a trade-off, BASE databases (yes, they are called BASE databases intentionally to mock ACID databases) use some tactics to have consistency, atomicity, and partition tolerance "eventually". They do not really defy the CAP theorem but work around it.

Simply put: I can afford my database to be consistent over time by synchronizing information between different database nodes. I can cache data (also called "soft-state") and persist it later to increase the response time of my database. I can have a number of database nodes with distributed data (partition tolerance) to be highly available and any loss of connectivity to any nodes prompts other nodes to take over!

This does not mean that BASE databases are not prone to failure. It does imply however, that they can recover quickly and consistently. They usually reside on standard commodity hardware, thus making them affordable for most businesses!

A lot of databases on websites prefer speed, performance, and scalability instead of pure consistency and integrity of data. However, as the next topic will cover, it is important to know what to choose!

Using ACID or BASE?

"**Atomic, Consistent, Isolated, and Durable**" (**ACID**) is a cliched term used for transactional databases. ACID databases are still very popular today but BASE databases are catching up.

ACID databases are good to use when you have heavy transactions at the core of your business processes. But most applications can live without this complexity. This does not imply that BASE databases do not support transactions, it's just that ACID databases are better suited for them.

Choose a database wisely – an old man said rightly! A choice of a database can decide the future of your product. There are many databases today that we can choose from. Here are some basic rules to help choose between databases for web applications:

- ◆ A large number of small writes (vote up/down) – Redis
- ◆ Auto-completion, caching – Redis, memcached
- ◆ Data mining, trending – MongoDB, Hadoop, and Big Table
- ◆ Content based web portals – MongoDB, Cassandra, and Sharded ACID databases
- ◆ Financial Portals – ACID database

Using Ruby

So, if you are now convinced (or rather interested to read on about MongoDB), you might wonder where Ruby fits in anyway? Ruby is one of the languages that is being adopted the fastest among all the new-age object oriented languages. But the big differentiator is that it is a language that can be used, tweaked, and cranked in any way that you want – from writing sweet smelling code to writing a **domain-specific language (DSL)**!

Ruby metaprogramming lets us easily adapt to any new technology, frameworks, API, and libraries. In fact, most new services today always bundle a Ruby gem for easy integration.

There are many Ruby implementations available today (sometimes called Rubies) such as, the original MRI, JRuby, Rubinius, MacRuby, MagLev, and the Ruby Enterprise Edition. Each of them has a slightly different flavors, much like the different flavors of Linux.

I often have to "sell" Ruby to nontechnical or technically biased people. This simple experiment never fails:

When I code in Ruby, I can guarantee, "My grandmother can read my code". Can any other language guarantee that? The following is a simple code in C:

```
/* A simple snippet of code in C */  
  
for (i = 0; i < 10; i++) {  
    printf("Hi");  
}
```

And now the same code in Ruby:

```
# The same snippet of code in Ruby  
  
10.times do  
    print "hi"  
end
```

There is no way that the Ruby code can be misinterpreted. Yes, I am not saying that you cannot write complex and complicated code in Ruby, but most code is simple to read and understand. Frameworks, such as Rails and Sinatra, use this feature to ensure that the code we see is readable! There is a lot of code under the cover which enables this though. For example, take a look at the following Ruby code:

```
# library.rb
class Library
    has_many :books
end

# book.rb
class Book
    belongs_to :library
end
```

It's quite understandable that "A library has many books" and that "A book belongs to a library".

The really fun part of working in Ruby (and Rails) is the finesse in the language. For example, in the small Rails code snippet we just saw, `books` is plural and `library` is singular. The framework infers the model `Book` model by the symbol `:books` and infers the `Library` model from the symbol `:library` – it goes the distance to make code readable.

As a language, Ruby is free flowing with relaxed rules – you can define a method call `true` in your calls that could return `false!` Ruby is a language where you do whatever you want as long as you know its impact. It's a human language and you can do the same thing in many different ways! There is no right or wrong way; there is only a more efficient way. Here is a simple example to demonstrate the power of Ruby! How do you calculate the sum of all the numbers in the array `[1, 2, 3, 4, 5]`?

The non-Ruby way of doing this in Ruby is:

```
sum = 0

for element in [1, 2, 3, 4, 5] do
    sum += element
end
```

The not-so-much-fun way of doing this in Ruby could be:

```
sum = 0

[1, 2, 3, 4, 5].each do |element|
    sum += element
end
```

The normal-fun way of doing this in Ruby is:

```
[1, 2, 3, 4, 5].inject(0) { |sum, element| sum + element }
```

Finally, the kick-ass way of doing this in Ruby is either one of the following:

```
[1, 2, 3, 4, 5].inject(&:+)  
[1, 2, 3, 4, 5].reduce(:+)
```

There you have it! So many different ways of doing the same thing in Ruby – but notice how most Ruby code gets done in one line.

Enjoy Ruby!

What this book covers

Chapter 1, Installing MongoDB and Ruby, describes how to install MongoDB on Linux and Mac OS. We shall learn about the various MongoDB utilities and their usage. We then install Ruby using RVM and also get a brief introduction to rbenv.

Chapter 2, Diving Deep into MongoDB, explains the various concepts of MongoDB and how it differs from relational databases. We learn various techniques, such as inserting and updating documents and searching for documents. We even get a brief introduction to Map/Reduce.

Chapter 3, MongoDB Internals, shares some details about what BSON is, usage of JavaScript, the global write lock, and why there are no joins or transactions supported in MongoDB. If you are a person in the fast lane, you can skip this chapter.

Chapter 4, Working Out Your Way with Queries, explains how we can query MongoDB documents and search inside different data types such as arrays, hashes, and embedded documents. We learn about the various query options and even regular expression based searching.

Chapter 5, Ruby DataMappers: Ruby and MongoDB Go Hand in Hand, provides details on how to use Ruby data mappers to query MongoDB. This is our first introduction to MongoMapper and Mongoid. We learn how to configure both of them, query using these data mappers, and even see some basic comparison between them.

Chapter 6, Modeling Ruby with Mongoid, introduces us to data models, Rails, Sinatra, and how we can model data using MongoDB data mappers. This is the core of the web application and we see various ways to model data, organize our code, and query using Mongoid.

Chapter 7, Achieving High Performance on Your Ruby Application with MongoDB, explains the importance of profiling and ensuring better performance right from the start of developing web applications using Ruby and MongoDB. We learn some best practices and concepts concerning the performance of web applications, tools, and methods which monitor the performance of our web application.

Chapter 8, Rack, Sinatra, Rails, and MongoDB – Making Use of them All, describes in detail how to build the full web application in Rails and Sinatra using Mongoid. We design the logical flow, the views, and even learn how to test our code and document it.

Chapter 9, Going Everywhere – Geospatial Indexing with MongoDB, helps us understand geolocation concepts. We learn how to set up geospatial indexes, get introduced to geocoding, and learn about geolocation spherical queries.

Chapter 10, Scaling MongoDB, provides details on how we scale MongoDB using replica sets. We learn about sharding, replication, and how we can improve performance using MongoDB map/reduce.

Appendix, Pop Quiz Answers, provides answers to the quizzes present at the end of chapters.

What you need for this book

This book would require the following:

- ◆ MongoDB version 2.0.2 or latest
- ◆ Ruby version 1.9 or latest
- ◆ RVM (for Linux and Mac OS only)
- ◆ DevKit (for Windows only)
- ◆ MongoMapper
- ◆ Mongoid

And other gems, of which I will inform you as we need them!

Who this book is for

This book assumes that you are experienced in Ruby and web development skills - HTML, and CSS. Having knowledge of using NoSQL will help you get through the concepts quicker, but it is not mandatory. No prior knowledge of MongoDB required.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

- 1.** Action 1
- 2.** Action 2
- 3.** Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
book = {  
    name: "Oliver Twist",  
    author: "Charles Dickens",  
    publisher: "Dover Publications",  
    published_on: "December 30, 2002",  
    category: ['Classics', 'Drama']  
}
```

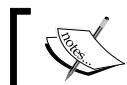
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function(key, values) {  
    var result = {votes: 0}  
  
    values.forEach(function(value) {  
        result.votes += value.votes;  
    });  
  
    return result;  
}
```

Any command-line input or output is written as follows:

```
$ curl -L get.rvm.io | bash -s stable
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Installing MongoDB and Ruby

MongoDB and Ruby have both been created as a result of technology getting complicated. They both try to keep it simple and manage all the complicated tasks at the same time. MongoDB manages "humongous" data and Ruby is fun. Working together, they form a great bond that gives us what most programmers desire—a fun way to build large applications!

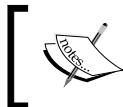
Now that your interest has increased, we should first set up our system. In this chapter, we will see how to do the following:

- ◆ Install Ruby using RVM
- ◆ Install MongoDB
- ◆ Configure MongoDB
- ◆ Set up the initial playground using MongoDB tools

But first, what are the basic system requirements for installing Ruby and MongoDB? Do we need a heavy-duty server? Nah! On the contrary, any standard workstation or laptop will be enough. Ensure that you have at least 1 GB memory and more than 32 GB disk space.

Did you say operating system? Ruby and MongoDB are both cross-platform compliant. This means they can work on any flavor of Linux (such as Ubuntu, Red Hat, Fedora, Gentoo, and SuSE), Mac OS (such as Leopard, Snow Leopard, and Lion) or Windows (such as XP, 2000, and 7).

If you are planning on using Ruby and MongoDB professionally, my personal recommendations for development are Mac OS or Linux. As we want to see detailed instructions, I am going to use examples for Ubuntu or Mac OS (and point out additional instructions for Windows whenever I can). While hosting MongoDB databases, I would personally recommend using Linux.



It's true that Ruby is cross-platform, most Rubyists tend to shy away from Windows as it's not always flawless. There are efforts underway to rectify this.



Let the games begin!

Installing Ruby

I recommend using **RVM (Ruby Version Manager)** for installing Ruby. The detailed instructions are available at <http://beginrescueend.com/rvm/install/>.



Incidentally, RVM was called Ruby Version Manager but its name was changed to reflect how much more it does today!



Using RVM on Linux or Mac OS

On Linux or Mac OS you can run this initial command to install RVM as follows:

```
$ curl -L get.rvm.io | bash -s stable  
$ source ~/.rvm/scripts/'rvm'
```

After this has been successfully run, you can verify it yourself.

```
$ rvm list known
```

If you have successfully installed RVM, this should show you the entire list of Rubies available. You will notice that there are quite a few implementations of Ruby (MRI Ruby, JRuby, Rubinius, REE, and so on) We are going to install MRI Ruby.



MRI Ruby is the "standard" or original Ruby implementation.
It's called Matz Ruby Interpreter.



The following is what you will see if you have successfully executed the previous command:

```
$ rvm list known  
# MRI Rubies  
[ruby-]1.8.6 [-p420]  
[ruby-]1.8.6-head  
[ruby-]1.8.7 [-p352]  
[ruby-]1.8.7-head  
[ruby-]1.9.1-p378  
[ruby-]1.9.1[-p431]  
[ruby-]1.9.1-head  
[ruby-]1.9.2-p180  
[ruby-]1.9.2[-p290]  
[ruby-]1.9.2-head  
[ruby-]1.9.3-preview1  
[ruby-]1.9.3-rc1  
[ruby-]1.9.3[-p0]  
[ruby-]1.9.3-head  
ruby-head  
  
# GoRuby  
goruby  
  
# JRuby  
jruby-1.2.0  
jruby-1.3.1  
jruby-1.4.0  
jruby-1.6.1  
jruby-1.6.2  
jruby-1.6.3  
jruby-1.6.4  
jruby[-1.6.5]  
jruby-head
```

```
# Rubinius
rbx-1.0.1
rbx-1.1.1
rbx-1.2.3
rbx-1.2.4
rbx[-head]
rbx-2.0.0pre

# Ruby Enterprise Edition
ree-1.8.6
ree[-1.8.7][-2011.03]
ree-1.8.6-head
ree-1.8.7-head

# Kiji
kiji

# MagLev
maglev[-26852]
maglev-head

# Mac OS X Snow Leopard Only
macruby[-0.10]
macruby-nightly
macruby-head

# IronRuby -- Not implemented yet.
ironruby-0.9.3
ironruby-1.0-rc2
ironruby-head
```

Isn't that beautiful? So many Rubies and counting!

**Fun fact**

Ruby is probably the only language that has a plural notation! When we work with multiple versions of Ruby, we collectively refer to them as "Rubies"!

Before we actually install any Rubies, we should configure the RVM packages that are necessary for all the Rubies. These are the standard packages that Ruby can integrate with, and we install them as follows:

```
$ rvm package install readline  
$ rvm package install iconv  
$ rvm package install zlib  
$ rvm package install openssl
```

The preceding commands install some useful libraries for all the Rubies that we will install. These libraries make it easier to work with the command line, internationalization, compression, and SSL. You can install these packages even after Ruby installation, but it's just easier to install them first.

```
$ rvm install 1.9.3
```

The preceding command will install Ruby 1.9.3 for us. However, while installing Ruby, we also want to pre-configure it with the packages that we have installed. So, here is how we do it, using the following commands:

```
$ export rvm_path=~/rvm  
$ rvm install 1.9.3 --with-readline-dir=$rvm_path/usr --with-iconv-dir=$rvm_path/usr --with-zlib-dir=$rvm_path/usr --with-openssl-dir=$rvm_path/usr
```

The preceding commands will miraculously install Ruby 1.9.3 configured with the packages we have installed. We should see something similar to the following on our screen:

```
$ rvm install 1.9.3
```

```
Installing Ruby from source to: /Users/user/.rvm/rubies/ruby-1.9.3-p0,  
this may take a while depending on your cpu(s)...
```

Installing MongoDB and Ruby

```
ruby-1.9.3-p0 - #fetching
ruby-1.9.3-p0 - #downloading
ruby-1.9.3-p0, this may take a while depending on your connection...

...
ruby-1.9.3-p0 - #extracting
ruby-1.9.3-p0 to /Users/user/.rvm/src/ruby-1.9.3-p0
ruby-1.9.3-p0 - #extracted to /Users/user/.rvm/src/ruby-1.9.3-p0
ruby-1.9.3-p0 - #configuring
ruby-1.9.3-p0 - #compiling
ruby-1.9.3-p0 - #installing

...
Install of ruby-1.9.3-p0 - #complete
```

Of course, whenever we start our machine, we do want to load RVM, so do add this line in your startup profile script:

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm" # Load RVM function' >> ~/.bash_profile
```

This will ensure that Ruby is loaded when you log in.



\$ rvm requirements is a command that can assist you on custom packages to be installed. This gives instructions based on the operating system you are on!

The RVM games

Configuring RVM for a project can be done as follows:

```
$ rvm -create -rvmrc use 1.9.3%myproject
```

The previous command allows us to configure a gemset for our project. So, when we move to this project, it has a `.rvmrc` file that gets loaded and voila — our very own custom workspace!

A **gemset**, as the name suggests, is a group of gems that are loaded for a particular version of Ruby or a project. As we can have multiple versions of the same gem on a machine, we can configure a gemset for a particular version of Ruby and for a particular version of the gem as well!

```
$ cd /path/to/myproject  
Using ruby 1.9.2 p180 with gemset myproject
```



In case you need to install something via RVM with sudo access, remember to use `rvm sudo` instead of sudo!

The Windows saga

RVM does not work on Windows, instead you can use pik. All the detailed instructions to install Ruby are available at <http://rubyinstaller.org/>. It is pretty simple and a one-click installer.



Do remember to install DevKit as it is required for compiling native gems.

Using rbenv for installing Ruby

Just like all good things, RVM becomes quite complex because the community started contributing heavily to it. Some people wanted just a Ruby version manager, so rbenv was born. Both are quite popular but there are quite a few differences between rbenv and RVM.

For starters, rbenv does not need to be loaded into the shell and does not override any shell commands. It's very lightweight and unobtrusive. Install it by cloning the repository into your home directory as `.rbenv`. It is done as follows:

```
$ cd  
$ git clone git://github.com/sstephenson/rbenv.git .rbenv
```

Add the preceding command to the system path, that is, the `$PATH` variable and you're all set.

rbenv works on a very simple concept of shims. **Shims** are scripts that understand what version of Ruby we are interested in. All the versions of Ruby should be kept in the `$HOME/.rbenv/versions` directory. Depending on which Ruby version is being used, the shim inserts that particular path at the start of the `$PATH` variable. This way, that Ruby version is picked up!

This enables us to compile the Ruby source code too (unlike RVM where we have to specify ruby-head).



For more information on rbenv, see <https://github.com/sstephenson/rbenv>.



Installing MongoDB

MongoDB installers are a bunch of binaries and libraries packaged in an archive. All you need to do is download and extract the archive. Could this be any simpler?

On Mac OS, you have two popular package managers Homebrew and MacPorts. If you are using Homebrew, just issue the following command:

```
$ brew install MongoDB
```

If you don't have brew installed, it is strongly recommended to install it. But don't fret. Here is the manual way to install MongoDB on any Linux, Mac OS, or Windows machine:

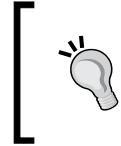
1. Download MongoDB from <http://www.mongodb.org/downloads>.
2. Extract the .tgz file to a folder (preferably which is in your system path).

It's done!

On any Linux Shell, you can issue the following commands to download and install. Be sure to append the /path/to/MongoDB/bin to your \$PATH variable:

```
$ cd /usr/local/  
$ curl http://fastdl.mongodb.org/linux/mongodb-linux-i686-2.0.2.tgz >  
mongo.tgz  
$ tar xf mongo.tgz  
$ ln -s mongodb-linux-i686-2.0.2 MongoDB
```

For Windows, you can simply download the ZIP file and extract it in a folder. Ensure that you update the </path/to/MongoDB/bin> in your system path.



MongoDB v1.6, v1.8, and v2.x are considerably different. Be sure to install the latest version. Over the course of writing this book, v2.0 was released and the latest version is v2.0.2. It is that version that this book will reference.



Configuring the MongoDB server

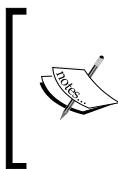
Before we start the MongoDB server, it's necessary to configure the path where we want to store our data, the interface to listen on, and so on. All these configurations are stored in `mongod.conf`. The default `mongod.conf` looks like the following code and is stored at the same location where MongoDB is installed—in our case `/usr/local/mongodb`:

```
# Store data in /usr/local/var/mongodb instead of the default /data/db
dbpath = /usr/local/var/mongodb

# Only accept local connections
bind_ip = 127.0.0.1
```

`dbpath` is the location where the data will be stored. Traditionally, this used to be `/data/db` but this has changed to `/usr/local/var/mongodb`. MongoDB will create this `dbpath` if you have not created it already.

`bind_ip` is the interface on which the server will run. Don't mess with this entry unless you know what you are doing!



Write-ahead logging is a technique to ensure durability and atomicity in database systems. Before actually writing to the database, the information (such as redo and undo) is written to a log (called the journal). This ensures that recovering from a crash is credible and fast. We shall learn more about this in the book.

Starting MongoDB

We can start the MongoDB server using the following command:

```
$ sudo mongod --config /usr/local/mongodb/mongod.conf
```

Remember that if we don't give the `--config` parameter, the default `dbpath` will be taken as `/data/db`.

When you start the server, if all is well, you should see something like the following:

```
$ sudo mongod --config /usr/local/mongodb/mongod.conf
```

```
Sat Sep 10 15:46:31 [initandlisten] MongoDB starting : pid=14914
port=27017 dbpath=/usr/local/var/mongodb 64-bit
```

```
Sat Sep 10 15:46:31 [initandlisten] db version v2.0.2, pdfile version 4.5

Sat Sep 10 15:46:31 [initandlisten] git version:
c206d77e94bc3b65c76681df5a6b605f68a2de05

Sat Sep 10 15:46:31 [initandlisten] build sys info: Darwin erh2.10gen.
cc 9.6.0 Darwin Kernel Version 9.6.0: Mon Nov 24 17:37:00 PST 2008;
root:xnu-1228.9.59~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_40

Sat Sep 10 15:46:31 [initandlisten] journal dir=/usr/local/var/mongodb/
journal

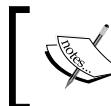
Sat Sep 10 15:46:31 [initandlisten] recover : no journal files present,
no recovery needed

Sat Sep 10 15:46:31 [initandlisten] waiting for connections on port 27017
```

```
Sat Sep 10 15:46:31 [websvr] web admin interface listening on port 28017
```

The preceding process does not terminate as it is running in the foreground! Some explanations are due here:

- ◆ The server started with pid 14914 on port 27017 (default port)
- ◆ The MongoDB version is 2.0.2
- ◆ The journal path is /usr/local/var/mongodb/journal (It also mentions that there is no current journal file, as this is the first time we are starting this up!)
- ◆ The web admin port is on 28017



The MongoDB server has some pretty interesting command-line options: -v is verbose. -vv is more verbose and -vvv is even more verbose. Include multiple times for more verbosity!

There are plenty of command line options that allow us to use MongoDB in various ways. For example:

1. --jsonp allows JSONP access.
2. --rest turns on REST API.
3. Master/Slave, options, replication options, and even sharing options (We shall see more in *Chapter 10, Scaling MongoDB*).

Stopping MongoDB

Press ***Ctrl+C*** if the process is running in the foreground. If it's running as a daemon, it has its standard startup script. On Linux flavors such as Ubuntu, you have upstart scripts that start and stop the `mongod` daemon. On Mac, you have `launchd` and `launchct` commands that can start and stop the daemon. On other flavors of Linux, you would find more of the resource scripts in the `/etc/init.d` directory. On Windows, the **Services** in the Control Panel can control the daemon process.

The MongoDB CLI

Along with the MongoDB server binary, there are plenty of other utilities too that help us in administration, monitoring, and management of MongoDB.

Understanding JavaScript Object Notation (JSON)

Even before we see how to use MongoDB utilities, it's important to know how information is stored. We shall study a lot more of the object model in *Chapter 2, Diving Deep into MongoDB*.

What is a JavaScript object? Surely you've heard of **JavaScript Object Notation (JSON)**. MongoDB stores information similar to this. (It's called **Binary JSON (BSON)**, which we shall read more about in *Chapter 3, The MongoDB Internals*). BSON, in addition to JSON formats, is ideally suited for "Document" storage. Don't worry, more information on this later!

So, if you want to save information, you simply use the JSON protocol:

```
{  
  name : 'Gautam Rege',  
  passion: [ 'Ruby', 'MongoDB' ],  
  company : {  
    name : "Josh Software Private Limited",  
    country : 'India'  
  }  
}
```

The previous example shows us how to store information:

```
String: "" or ''  
Integer: 10  
Float: 10.1  
Array: ['1', 2]  
Hash: {a: 1, b: 2}
```

Connecting to MongoDB using Mongo

The Mongo client utility is used to connect to MongoDB database. Considering that this is a Ruby and MongoDB book, it is a utility that we shall use rarely (because we shall be accessing the database using Ruby). The Mongo CLI client, however, is indeed useful for testing out basics.

We can connect to MongoDB databases in various ways:

```
$ mongo book
$ mongo 192.168.1.100/book
$ mongo db.myserver.com/book
$ mongo 192.168.1.100:9999/book
```

In the preceding case, we connect to a database called `book` on localhost, on a remote server, or on a remote server on a different port. When you connect to a database, you should see the following:

```
$ mongo book
MongoDB shell version: 2.0.2
connecting to: book
>
```

Saving information

To save data, use the JavaScript object and execute the following command:

```
> db.shelf.save( { name: 'Gautam Rege',
    passion : [ 'Ruby', 'MongoDB' ]
})
>
```

The previous command saves the data (that is, usually called "Document") into the collection `shelf`. We shall talk more about collections and other terminologies in *Chapter 3, MongoDB Internals*. A collection can vaguely be compared to tables.

Retrieving information

We have various ways to retrieve the previously stored information:

- ◆ Fetch the first 10 objects from the book database (also called a collection), as follows:

```
> db.shelf.find()
{ "_id" : ObjectId("4e6bb98a26e77d64db8a3e89") , "name" : "Gautam
Rege" , "passion" : [ "Ruby" , MongoDB" ] }
>
```

- ◆ Find a specific record of the name attribute. This is achieved by executing the following command:

```
> db.shelf.find( { name : 'Gautam Rege' } )
{ "_id" : ObjectId("4e6bb98a26e77d64db8a3e89") , "name" : "Gautam
Rege" , "passion" : [ "Ruby" , MongoDB" ] }
>
```

So far so good! But you may be wondering what the big deal is. This is similar to a select query I would have fired anyway. Well, here is where things start getting interesting.

- ◆ Find records by using regular expressions! This is achieved by executing the following command:

```
$ db.shelf.find( { name : /Rege/ } )
{ "_id" : ObjectId("4e6bb98a26e77d64db8a3e89") , "name" : "Gautam
Rege" , "passion" : [ "Ruby" , MongoDB" ] }
>
```

- ◆ Find records by using regular expressions using the case-insensitive flag! This is achieved by executing the following command:

```
$ db.shelf.find( { name : /rege/i } )
{ "_id" : ObjectId("4e6bb98a26e77d64db8a3e89") , "name" : "Gautam
Rege" , "passion" : [ "Ruby" , MongoDB" ] }
>
```

As we can see, it's easy when we have programming constructs mixed with database constructs with a dash of regular expressions.

Deleting information

No surprises here!

- ◆ To remove all the data from book, execute the following command:

```
> db.shelf.remove()  
>
```

- ◆ To remove specific data from book, execute the following command:

```
> db.shelf.remove({name : 'Gautam Rege'})  
>
```

Exporting information using mongoexport

Ever wondered how to extract information from MongoDB? It's `mongoexport`! What is pretty cool is that the Mongo data transfer protocol is all in JSON/BSON formats. So what? - you ask. As JSON is now a universally accepted and common format of data transfer, you can actually export the database, or the collection, directly in JSON format — so even your web browser can process data from MongoDB. No more three-tier applications! The opportunities are infinite!

Ok, back to basics. Here is how you can export data from MongoDB:

```
$ mongoexport -d book -c shelf  
connected to: 127.0.0.1  
{ "_id" : { "$oid" : "4e6c45b81cb76a67a0363451" }, "name" : "Gautam  
Rege", "passion" : [ "Ruby", "MongoDB" ] }  
exported 1 records
```

This couldn't be simpler, could it? But wait, there's more. You can export this data into a CSV file too!

```
$ mongoexport -d book -c shelf -f name,passion --csv -o test.csv
```

The preceding command saves data in a CSV file. Similarly, you can export data as a JSON array too!

```
$ mongoexport -d book -c shelf --jsonArray  
connected to: 127.0.0.1  
[ { "_id" : { "$oid" : "4e6c61a05ff70cac810c6996" }, "name" : "Gautam  
Rege", "passion" : [ "Ruby", "MongoDB" ] } ]  
exported 1 records
```

Importing data using mongoimport

Wasn't this expected? If there is a `mongoexport`, you must have a `mongoimport`! Imagine when you want to import information; you can do so in a JSON array, CSV, TSV or plain JSON format. Simple and sweet!

Managing backup and restore using mongodump and mongorestore

Backups are important for any database and MongoDB is no exception. `mongodump` dumps the entire database or databases in binary JSON format. We can store this and use this later to restore it from the backup. This is the closest resemblance to `mysqldump`! It is done as follows:

```
$ mongodump -dconfig
connected to: 127.0.0.1
DATABASE: config to dump/config
    config.version to dump/config/version.bson
        1 objects
    config.system.indexes to dump/config/system.indexes.bson
        14 objects
...
    config.collections to dump/config/collections.bson
        1 objects
    config.changelog to dump/config/changelog.bson
        10 objects
$

$ ls dump/config/
changelog.bson     databases.bson     mongos.bson     system.indexes.bson
chunks.bson        lockpings.bson    settings.bson   version.bson
collections.bson   locks.bson      shards.bson
```

Now that we have backed up the database, in case we need to restore it, it is just a matter of supplying the information to `mongorestore`, which is done as follows:

```
$ mongorestore -dbkp1 dump/config/
connected to: 127.0.0.1
dump/config/changelog.bson
```

```
going into namespace [bkp1.changelog]
10 objects found
dump/config/chunks.bson
going into namespace [bkp1.chunks]
7 objects found
dump/config/collections.bson
    going into namespace [bkp1.collections]
1 objects found
dump/config/databases.bson
    going into namespace [bkp1.databases]
15 objects found
dump/config/lockpings.bson
    going into namespace [bkp1.lockpings]
5 objects found
...
1 objects found
dump/config/system.indexes.bson
    going into namespace [bkp1.system.indexes]
{ key: { _id: 1 }, ns: "bkp1.version", name: "_id_" }
{ key: { _id: 1 }, ns: "bkp1.settings", name: "_id_" }
{ key: { _id: 1 }, ns: "bkp1.chunks", name: "_id_" }
{ key: { ns: 1, min: 1 }, unique: true, ns: "bkp1.chunks", name: "ns_1_min_1" }
...
{ key: { _id: 1 }, ns: "bkp1.databases", name: "_id_" }
{ key: { _id: 1 }, ns: "bkp1.collections", name: "_id_" }
14 objects found
```

Saving large files using mongofiles

The database should be able to store a large amount of data. Typically, the maximum size of JSON objects stores 4 MB (and in v1.7 onwards, 16 MB). So, can we store videos and other large documents in MongoDB? That is where the `mongofiles` utility helps.

MongoDB uses GridFS specification for storing large files. Language bindings are available to store large files. GridFS splits larger files into chunks and maintains all the metadata in the collection. It's interesting to note that GridFS is just a specification, not a mandate and all MongoDB drivers adhere to this voluntarily.

To manage large files directly in a database, we use the `mongofiles` utility.

```
$ mongofiles -d book -c shelf put /home/gautam/Relax.mov

connected to: 127.0.0.1

added file: { _id: ObjectId('4e6c6f9cc7bd0bf42f31aa3b'), filename:
"/Users/gautam/Relax.mov", chunkSize: 262144, uploadDate: new
Date(1315729317190), md5: "43883ace6022c8c6682881b55e26e745", length:
49120795 }

done!
```

Notice that 47 MB of data was saved in the database. I wouldn't want to leave you in the dark, so here goes a little bit of explanation. GridFS creates an `fs` collection that has two more collections called `chunks` and `files`. You can retrieve this information from MongoDB from the command line or using Mongo CLI.

```
$ mongofiles -d book list
connected to: 127.0.0.1
/Users/gautam/Downloads/Relax.mov 49120795
```

Let's use Mongo CLI to fetch this information now. This can be done as follows:

```
$ mongo
MongoDB shell version: 1.8.3
connecting to: test
> use book
switched to db book
> db.fs.chunks.count()
188
> db.fs.files.count()
1
> db.fs.files.findOne()
{
    "_id" : ObjectId("4e6c6f9cc7bd0bf42f31aa3b"),
    "filename" : "/Users/gautam/Downloads/Relax.mov",
    "chunkSize" : 262144,
```

```
"uploadDate" : ISODate("2011-09-11T08:21:57.190Z"),
"md5" : "43883ace6022c8c6682881b55e26e745",
"length" : 49120795
}
>
```

bsondump

This is a utility that helps analyze BSON dumps. For example, if you want to filter all the objects from a BSON dump of the book database, you could run the following command:

```
$ bsondump --filter "{name:/Rege/}" dump/book/shelf.bson
```

This command would analyze the entire dump and get all the objects where name has the specified value in it! The other very nice feature of bsondump is if we have a corrupted dump during any restore, we can use the objcheck flag to ignore all the corrupt objects.

Installing Rails/Sinatra

Considering that we aim to do web development with Ruby and MongoDB, Rails or Sinatra cannot be far behind.



Rails 3 packs a punch. Sinatra was born because Rails 2.x was a really heavy framework. However, Rails 3 has Metal that can be configured with only what we need in our application framework. So Rails 3 can be as lightweight as Sinatra and also get the best of the support libraries. So Rails 3 it is, if I have to choose between Ruby web frameworks!

Installing Rails 3 or Sinatra is as simple as one command, as follows:

```
$ gem install rails
$ gem install sinatra
```



At the time of writing this chapter, Rails 3.2 had just been released in production mode. That is what we shall use!

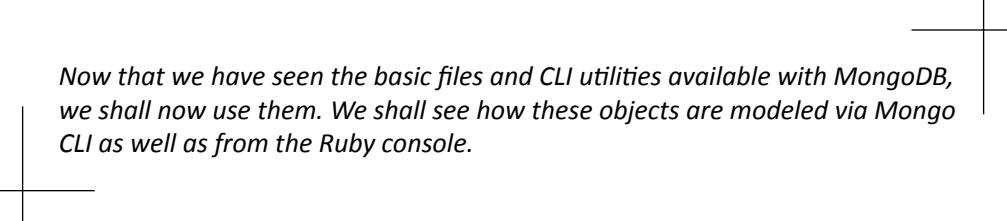
Summary

What we have learned so far is about getting comfortable with Ruby and MongoDB. We installed Ruby using RVM, learned a little about rbenv and then installed MongoDB. We saw how to configure MongoDB, start it, stop it, and finally we played around with the various MongoDB utilities to dump information, restore it, save large files and even export to CSV or JSON.

In the next chapter, we shall dive deep into MongoDB. We shall learn how to work with documents, save them, fetch them, and search for them — all this using the mongo utility. We shall also see a comparison with SQL databases.

2

Diving Deep into MongoDB



Now that we have seen the basic files and CLI utilities available with MongoDB, we shall now use them. We shall see how these objects are modeled via Mongo CLI as well as from the Ruby console.

In this chapter we shall learn the following:

- ◆ Modeling the application data.
- ◆ Mapping it to MongoDB objects.
- ◆ Creating embedded and relational objects.
- ◆ Fetching objects.
- ◆ How does this differ from the SQL way?
- ◆ Take a brief look at a Map/Reduce, with an example.

We shall start modeling an application, whereby we shall learn various constructs of MongoDB and then integrate it into Rails and Sinatra. We are going to build the Sodibee (pronounced as |saw-d-bee|) Library Manager.

Books belong to particular categories including Fiction, Non-fiction, Romance, Self-learning, and so on. Books belong to an author and have one publisher.

Books can be leased or bought. When books are bought or leased, the customer's details (such as name, address, phone, and e-mail) are registered along with the list of books purchased or leased.

An inventory maintains the quantity of each book with the library, the quantity sold and the number of times it was leased.

Over the course of this book, we shall evolve this application into a full-fledged web application powered by Ruby and MongoDB. In this chapter we will learn the various constructs of MongoDB.

Creating documents

Let's first see how we can create documents in MongoDB. As we have briefly seen, MongoDB deals with collections and documents instead of tables and rows.

Time for action – creating our first document

Suppose we want to create the `book` object having the following schema:

```
book = {  
    name: "Oliver Twist",  
    author: "Charles Dickens",  
    publisher: "Dover Publications",  
    published_on: "December 30, 2002",  
    category: ['Classics', 'Drama']  
}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

On the Mongo CLI, we can add this `book` object to our collection using the following command:

```
> db.books.insert(book)
```

Suppose we also add the `shelf` collection (for example, the floor, the row, the column the shelf is in, the book indexes it maintains, and so on that are part of the `shelf` object), which has the following structure:

```
shelf : {  
    name : 'Fiction',  
    location : { row : 10, column : 3 },  
    floor : 1  
    lex : { start : 'O', end : 'P' }  
}
```

Remember, it's quite possible that a few years down the line, some shelf instances may become obsolete and we might want to maintain their record. Maybe we could have another shelf instance containing only books that are to be recycled or donated. What can we do? We can approach this as follows:

- ◆ **The SQL way:** Add additional columns to the table and ensure that there is a default value set in them. This adds a lot of redundancy to the data. This also reduces the performance a little and considerably increases the storage. Sad but true!
- ◆ **The NoSQL way:** Add the additional fields whenever you want. The following are the MongoDB schemaless object model instances:

```
> db.book.shelf.find()  
{ "_id" : ObjectId("4e81e0c3eeef2ac76347a01c"), "name" : "Fiction",  
  "location" : { "row" : 10, "column" : 3 }, "floor" : 1 }  
{ "_id" : ObjectId("4e81e0fdeeff2ac76347a01d"), "name" : "Romance",  
  "location" : { "row" : 8, "column" : 5 }, "state" : "window broken",  
  "comments" : "keep away from children" }
```

What just happened?

You will notice that the second object has more fields, namely comments and state. When fetching objects, it's fine if you get extra data. That is the beauty of NoSQL. When the first document is fetched (the one with the name Fiction), it will not contain the state and comments fields but the second document (the one with the name Romance) will have them.

Are you worried what will happen if we try to access non-existing data from an object, for example, accessing comments from the first object fetched? This can be logically resolved—we can check the existence of a key, or default to a value in case it's not there, or ignore its absence. This is typically done anyway in code when we access objects.

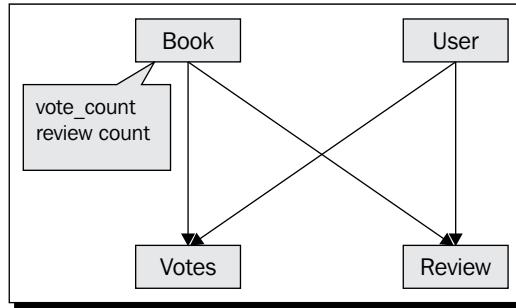
Notice that when the schema changed we did not have to add fields in every object with default values like we do when using a SQL database. So there is no redundant information in our database. This ensures that the storage is minimal and in turn the object information fetched will have concise data. So there was no redundancy and no compromise on storage or performance. But wait! There's more.

NoSQL scores over SQL databases

The way many-to-many relations are managed tells us how we can do more with MongoDB that just cannot be simply done in a relational database. The following is an example:

Each book can have reviews and votes given by customers. We should be able to see these reviews and votes and also maintain a list of top voted books.

If we had to do this in a relational database, this would be somewhat like the relationship diagram shown as follows: (get scared now!)



The `vote_count` and `review_count` fields are inside the `books` table that would need to be updated every time a user votes up/down a book or writes a review. So, to fetch a book along with its votes and reviews, we would need to fire three queries to fetch the information:

```
SELECT * from book where id = 3;  
SELECT * from reviews where book_id = 3;  
SELECT * from votes where book_id = 3;
```

We could also use a join for this:

```
SELECT * FROM books JOIN reviews ON reviews.book_id = books.id JOIN votes  
ON votes.book_id = books.id;
```



In MongoDB, we can do this directly using embedded documents or relational documents.



Using MongoDB embedded documents

Embedded documents, as the name suggests, are documents that are embedded in other documents. This is one of the features of MongoDB and this cannot be done in relational databases. Ever heard of a table embedded inside another table?

Instead of four tables and a complex many-to-many relationship, we can say that reviews and votes are part of a book. So, when we fetch a book, the reviews and the votes automatically come along with the book.

Embedded documents are analogous to chapters inside a book. Chapters cannot be read unless you open the book. Similarly embedded documents cannot be accessed unless you access the document.



For the UML savvy, embedded documents are similar to the contains or composition relationship.



Time for action – embedding reviews and votes

In MongoDB, the embedded object physically resides inside the parent. So if we had to maintain reviews and votes we could model the object as follows:

```
book : { name: "Oliver Twist",

  reviews : [
    { user: "Gautam",
      comment: "Very interesting read"
    },
    { user: "Harry",
      comment: "Who is Oliver Twist?"
    }
  ]
  votes: [ "Gautam", "Tom", "Dick"]
}
```

What just happened?

We now have reviews and votes inside the book. They cannot exist on their own. Did you notice that they look similar to JSON hashes and arrays? Indeed, they are an array of hashes. Embedded documents are just like hashes inside another object.

There is a subtle difference between hashes and embedded objects as we shall see later on in the book.

Have a go hero – adding more embedded objects to the book

Try to add more embedded objects such as `orders` inside the `book` document. It works!

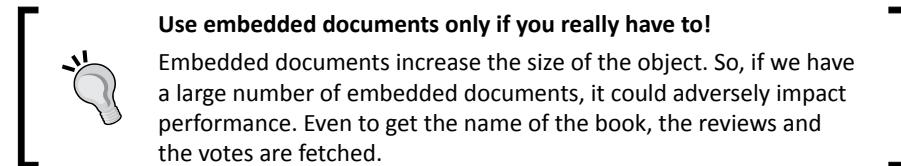
```
order = {
  name: "Toby Jones"
  type: "lease",
  units: 1,
  cost: 40
}
```

Fetching embedded objects

We can fetch a book along with the reviews and the votes with it. This can be done by executing the following command:

```
> var book = db.books.findOne({name : 'Oliver Twist'})  
> book.reviews.length  
2  
> book.votes.length  
3  
> book.reviews  
[  
  { user: "Gautam",  
    comment: "Very interesting read"  
  },  
  { user: "Harry",  
    comment: "Who is Oliver Twist?"  
  }  
]  
  
> book.votes  
[ "Gautam", "Tom", "Dick" ]
```

This does indeed look simple, doesn't it? By fetching a single object, we are able to get the review and vote count along with the data.



Using MongoDB document relationships

Just like we have embedded documents, we can also set up relationships between different documents.

Time for action – creating document relations

The following is another way to create the same relationship between books, users, reviews, and votes. This is more like the SQL way.

```
book: {
  _id: ObjectId("4e81b95ffed0eb0c23000002"),
  name: "Oliver Twist",
  author: "Charles Dickens",
  publisher: "Dover Publications",
  published_on: "December 30, 2002",
  category: ['Classics', 'Drama']
}
```

 Every document that is created in MongoDB has an object ID associated with it. In the next chapter, we shall soon learn about object IDs in MongoDB. By using these object IDs we can easily identify different documents. They can be considered as primary keys.

So, we can also create the `reviews` collection and the `votes` collection as follows:

```
users: [
  {
    _id: ObjectId("8d83b612fed0eb0bee000702"),
    name: "Gautam"
  },
  {
    _id : ObjectId("ab93b612fed0eb0bee000883"),
    name: "Harry"
  }
]

reviews: [
  {
    _id: ObjectId("5e85b612fed0eb0bee000001"),
    user_id: ObjectId("8d83b612fed0eb0bee000702"),
    book_id: ObjectId("4e81b95ffed0eb0c23000002"),
    comment: "Very interesting read"
  },
  {
    _id: ObjectId("4585b612fed0eb0bee000003"),
    user_id : ObjectId("ab93b612fed0eb0bee000883"),
    book_id: ObjectId("4e81b95ffed0eb0c23000002"),
  }
]
```

```
        comment: "Who is Oliver Twist?"  
    }  
]  
  
votes: [  
{  
    _id: ObjectId("6e95b612fed0eb0bee000123") ,  
    user_id : ObjectId("8d83b612fed0eb0bee000702") ,  
    book_id: ObjectId("4e81b95ffed0eb0c23000002") ,  
},  
{  
    _id: ObjectId("4585b612fed0eb0bee000003") ,  
    user_id : ObjectId("ab93b612fed0eb0bee000883") ,  
}  
]
```

What just happened?

Hmm!! Not very interesting, is it? It doesn't even seem right. That's because it isn't the right choice in this context. It's very important to know how to choose between nesting documents and relating them.



In your object model, if you will never search by the nested document (that is, look up for the parent from the child), embed it.

Just in case you are not sure about whether you would need to search by an embedded document, don't worry too much – it does not mean that you cannot search among embedded objects. You can use Map/Reduce to gather the information. There is more on this later in this chapter and a lot more in detail, in *Chapter 4, Working out Your Way with Queries*.

Comparing MongoDB versus SQL syntax

This is a good time to sit back and evaluate the similarities and dissimilarities between the MongoDB syntax and the SQL syntax. Let's map them together:

SQL commands	NoSQL (MongoDB) equivalent
SELECT * FROM books	db.books.find()
SELECT * FROM books WHERE id = 3;	db.books.find({ id : 3 })

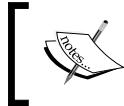
SQL commands	NoSQL (MongoDB) equivalent
SELECT * FROM books WHERE name LIKE 'Oliver%'	db.books.find({ name : /^Oliver/ })
SELECT * FROM books WHERE name like '%Oliver%'	db.books.find({ name : /Oliver/ })
SELECT * FROM books WHERE publisher = 'Dover Publications' AND published_date = "2011-8-01"	db.books.find({ publisher : "Dover Publications", published_date : ISODate("2011-8-01") })
SELECT * FROM books WHERE published_date > "2011-8-01"	db.books.find ({ published_date : { \$gt : ISODate("2011-8-01") } })
SELECT name FROM books ORDER BY published_date	db.books.find({}, { name : 1 }).sort({ published_date : 1 })
SELECT name FROM books ORDER BY published_date DESC	db.books.find({}, { name : 1 }).sort({ published_date : -1 })
SELECT votes.name from books JOIN votes where votes.book_id = books.id	db.books.find({ votes : { \$exists : 1 } }, { votes.name : 1 })

Some more notable comparisons between MongoDB and relational databases are:

- ◆ MongoDB does not support joins. Instead it fires multiple queries or uses Map/Reduce. We shall soon see why the NoSQL faction does not favor joins.
- ◆ SQL has stored procedures. MongoDB supports JavaScript functions.
- ◆ MongoDB has indexes similar to SQL.
- ◆ MongoDB also supports Map/Reduce functionality.
- ◆ MongoDB supports atomic updates like SQL databases.
- ◆ Embedded or related objects are used sometimes instead of a SQL join.
- ◆ MongoDB collections are analogous to SQL tables.
- ◆ MongoDB documents are analogous to SQL rows.

Using Map/Reduce instead of join

We have seen this mentioned a few times earlier—it's worth jumping into it, at least briefly.



Map/Reduce is a concept that was introduced by Google in 2004. It's a way of distributed task processing. We "map" tasks to works and then "reduce" the results.



Understanding functional programming

Functional programming is a programming paradigm that has its roots from lambda calculus. If that sounds intimidating, remember that JavaScript could be considered a functional language. The following is a snippet of functional programming:

```
$ (document) . ready( function () {
    $('#element') . click( function () {
        # do something here
    )) ;

    $('#element2') . change( function () {
        # do something here
    )) ;

});
```

We can have functions inside functions. Higher-level languages (such as Java and Ruby) support anonymous functions and closures but are still procedural functions. Functional programs rely on results of a function being chained to other functions.

Building the map function

The map function processes a chunk of data. Data that is fed to this function could be accessed across a distributed filesystem, multiple databases, the Internet, or even any mathematical computation series!

```
function map(void) -> void
```

The map function "emits" information that is collected by the "mystical super gigantic computer program" and feeds that to the reducer functions as input.

MongoDB as a database supports this paradigm making it "the all powerful" (of course I am joking, but it does indeed make MongoDB very powerful).

Time for action – writing the map function for calculating vote statistics

Let's assume we have a document structure as follows:

```
{
    name: "Oliver Twist",
    votes: ['Gautam', 'Harry']
    published_on: "December 30, 2002"
}
```

The map function for such a structure could be as follows:

```
function() {
    emit( this.name, {votes : this.votes} );
}
```

What just happened?

The `emit` function emits the data. Notice that the data is emitted as a (key, value) structure.

- ◆ **Key:** This is the parameter over which we want to gather information. Typically it would be some primary key, or some key that helps identify the information.



For the SQL savvy, typically the key is the field we use in the `GROUP BY` clause.

- ◆ **Value:** This is a JSON object. This can have multiple values and this is the data that is processed by the reduce function.

We can call `emit` more than once in the map function. This would mean we are processing data multiple times for the same object.

Building the reduce function

The reduce functions are the consumer functions that process the information emitted from the map functions and emit the results to be aggregated. For each emitted data from the map function, a reduce function emits the result. MongoDB collects and collates the results. This makes the system of collection and processing as a massive parallel processing system giving the all mighty power to MongoDB.

The reduce functions have the following signature:

```
function reduce(key, values_array) -> value
```

Time for action – writing the reduce function to process emitted information

This could be the reduce function for the previous example:

```
function(key, values) {  
    var result = {votes: 0}  
  
    values.forEach(function(value) {  
        result.votes += value.votes;  
    });  
  
    return result;  
}
```

What just happened?

reduce takes an array of values – so it is important to process an array every time. Later on in the book we shall see how there are various options to Map/Reduce that help us process data.

Let's analyze this function in more detail:

```
function(key, values) {  
    var result = {votes: 0}  
  
    values.forEach(function(value) {  
        result.votes += value.votes;  
    });  
  
    return result;  
}
```

The variable `result` has a structure similar to what was emitted from the map function. This is important, as we want the results from every document in the same format. If we need to process more results, we can use the `finalize` function (more on that later). The `result` function has the following structure:

```
function(key, values) {  
    var result = {votes: 0}  
  
    values.forEach(function(value) {  
        result.votes += value.votes;  
    });  
  
    return result;  
}
```

The values are always passed as arrays. It's important that we iterate the array, as there could be multiple values emitted from different map functions with the same key. So, we processed the array to ensure that we don't overwrite the results and collate them.

Understanding the Ruby perspective

Until now we have just been playing around with MongoDB. Now let's have a look at this from Ruby. Aaahhh... bliss!

For this example, we shall write some basic classes in Ruby. We are using Rails 3 and the Mongoid wrapper for MongoDB. (We shall see more about MongoDB wrappers later in the book)

Setting up Rails and MongoDB

To set up a Rails project, we first need to install the Rails gem. We shall also install the Bundler gem that goes hand-in-hand with Rails.

Time for action – creating the project

First we shall create the sample Rails project. Assuming you have installed Ruby already, we need to install Rails. The following command shows how to install Rails and Bundler.

```
$ gem install rails  
$ gem install bundler
```

What just happened?

The preceding commands will install Rails and Bundler. For the sake of this example, I am working with Rails 3.2.0 (that is, the current latest version) but I recommend that you should use the latest version of Rails available.

Understanding the Rails basics

Rails is a web framework written in Ruby. It was released publicly in 2005 and it has gathered a lot of steam since then. It is interesting to note that until Rails 2.x, the framework was a tightly coupled one. This was when other loosely coupled web frameworks made their way into the developer market. The most popular among them were Merb and Sinatra. These frameworks leveraged Ruby to its full potential but were competing against each other.



Around 2008-2009, the Rails core team (David Hanson and team) met the makers of Merb (Yehuda Katz and team) and they got together and discussed a strategy that has literally changed the face of web development. Rails 3 emerged with a bang; it had a brand new framework with Metal and Rack with loosely coupled components and very customizable middleware. This has made Rails extremely popular today.

Using Bundler

Bundler is another awesome gem by "Carlhuda" (Yahuda and Carl Leche) that manages gem dependencies in Ruby applications.

Why do we need the Bundler

In the "olden" days, when everything was a system installation, things would be running smoothly till somebody upgraded a system library or a gem... and then Kaboom! – the application crashed for no apparent reason and no code change. Some libraries break compatibility, which in turn requires us to install the new gems. So, even if a system administrator upgraded the system (as a routine maintenance activity), our Ruby application was prone to crashes.

A bigger problem arose when we were required to install multiple Ruby applications on the same system. Ruby version, Rails version, gem versions, and system libraries all could potentially clash to make development and deployment a nightmare!

One solution was to freeze gems and the Ruby version. This required us to ship everything into our application bundle. Not only was this inefficient but also increased the size of the bundle.

Then came along Bundler and, as the name suggests, it keeps track of dependencies in a Ruby application. Java has a similar package called Maven. But wait! Bundler has more in store. We can now package gems (via a Gemfile) and specify environments with it. So, if we require some gems only for testing, it can be specified to be a part of only the "test" group.

If that's not sold you over using Bundler, we can specify the source of the gem files too – github, sourceforge or even a gem in our local file system.

Bundler generates `Gemfile.lock` that manages the gem dependencies for the application. It uses the system-installed gems; so that we don't have to freeze gems or Ruby versions with each application.

Setting up Sodibee

Now that we have installed Rails and Bundler, it's time to set up the Sodibee project.

Time for action – start your engines

Now we shall create the Sodibee project in Rails 3. It can be done using the following command:

```
$ rails new sodibee -J-O
```

In the previous command, `-J` means `skip-prototype` (and use jQuery instead) and `-O` means `skip-activerecord`. This is important, as we want to use MongoDB.

Add the following to `Gemfile`:

```
gem 'mongoid'  
gem 'bson'  
gem 'bson_ext'
```

Now on command line, type the following:

```
$ bundle install
```



In Rails 3.2.1 a lot of automation has been added. `bundle install` is part of the process of creating a project.



What just happened?

The previous command: `bundle install` fetches missing gems, their dependencies, and installs them. It then generates `Gemfile.lock`. After `bundle install` is complete, you would see the following on the screen:

```
$ bundle install  
Fetching source index for http://rubygems.org/  
Using rake (0.9.2)  
Using abstract (1.0.0)
```

```
Using activesupport (3.2.0)
Using builder (2.1.2)
Using i18n (0.5.0)
Using activemodel (3.2.0)
Using erubis (2.6.6)
Using rack (1.2.4)
Using rack-mount (0.6.14)
Using rack-test (0.5.7)
Installing tzinfo (0.3.30)
Using actionpack (3.2.0)
Using mime-types (1.16)
Using polyglot (0.3.2)
Using treetop (1.4.10)
Using mail (2.2.19)
Using actionmailer (3.2.0)
Using arel (2.0.10)
Using activerecord (3.2.0)
Using activeresource (3.2.0)
Using bson (1.4.0)
Using bundler (1.0.10)
Using mongo (1.3.1)
Installing mongoid (2.2.1)
Using rdoc (3.9.4)
Using thor (0.14.6)
Using railties (3.2.0)
Using rails (3.2.0)
Your bundle is complete! Use `bundle show [gemname]` to see where a
bundled gem is installed.
```

Setting up Mongoid

Now that the Rails application is set up, let's configure Mongoid.

Mongoid is an **Object Document Mapper (ODM)** tool that maps Ruby objects to MongoDB documents. We shall learn a lot more in detail in the later chapters on Mongoid and other similar ODM tools. For now, we shall simply issue the command to configure Mongoid.

Time for action – configuring Mongoid

The Mongoid gem has a Rails generator command to configure Mongoid.



A Rails generator, as the name suggests, sets up files. Generators are used frequently in gems to set up config files, with default settings, `g` can be used instead of writing `generate`.

```
$ rails g mongoid:config
```

What just happened?

This command created a `config/mongoid.yml` file that is used to connect to MongoDB. The file would look like the following code snippet:

```
development:
  host: localhost
  database: sodibee_development

test:
  host: localhost
  database: sodibee_test

# set these environment variables on your prod server
production:
  host: <%= ENV['MONGOID_HOST'] %>
  port: <%= ENV['MONGOID_PORT'] %>
  username: <%= ENV['MONGOID_USERNAME'] %>
  password: <%= ENV['MONGOID_PASSWORD'] %>
  database: <%= ENV['MONGOID_DATABASE'] %>
  # slaves:
  #   - host: slave1.local
  #     port: 27018
  #   - host: slave2.local
  #     port: 27019
gautam-2:sodibee gautam$
```

Notice that there are now three environments to work with—development, test, and production. By default, Rails will pick up the development environment. We do not need to explicitly create the database in MongoDB. The first call to the database will create the database for us.

The previous command also configures the `config/application.rb` to ensure that ActiveRecord is disabled. ActiveRecord is the default Rails **ORM (Object Relational Mapper)**. As we are using Mongoid, we need to disable ActiveRecord.

Building the models

Now that we have the project set up, it's time we create the models. Each model will autogenerate collections in MongoDB. To create a model, all we need to do is create a file in the `app/models` folder.

Time for action – planning the object schema

Here we shall build the different models and add their relations.

Building the book model

This `app/models/book.rb` would contain the following code:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

  field :votes, type: Array

  belongs_to :author
  has_and_belongs_to_many :categories

  embeds_many :reviews
end
```

What just happened?

Let's study the previous code snippet in more detail:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date
```

```

    field :votes, type: Array

    belongs_to :author
    has_and_belongs_to_many :categories

    embeds_many :reviews
end

```

The preceding code includes the Mongoid module to save the documents in MongoDB.

 include is the Ruby way of adding methods to the Ruby class by including modules. This is called **module mixin**. We can include as many modules in a class as we want. Modules make the class richer by adding all the module methods as instance methods.
extend is the Ruby way of adding class methods to a Ruby class by including modules in it. All the methods from the modules included become class methods.

Let's have a look at the previous snippet again:

```

class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

  field :votes, type: Array

  belongs_to :author
  has_and_belongs_to_many :categories

  embeds_many :reviews
end

```

The previous code configures the name and the type of the fields for a document.

 Notice the Ruby 1.9 syntax for a hash. No more hash rockets (`=>`). Instead in we use the JSON notation directly. Remember it's `type: String` and not `type : String`. You must have the key and the colon (:) together.

Let's have a look at the snippet again:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

  field :votes, type: Array

  belongs_to :author

  has_and_belongs_to_many :categories

  embeds_many :reviews
end
```

The previous snippet is a relational document. This means that the document has a reference to the `author` document.

Let's have a look at the snippet for the second time:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

  field :votes, type: Array

  belongs_to :author
  has_and_belongs_to_many :categories

  embeds_many :reviews
end
```

The previous snippet is a many-to-many relationship between books and categories.

Let's have a look at the snippet a third time:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
```

```
field :published_on, type: Date  
field :votes, type: Array  
belongs_to :author  
has_and_belongs_to_many :categories  
embeds_many :reviews  
end
```

The previous snippet is an example of nested or embedded documents. All the review documents will be embedded into the books.

Have a go hero – building the remaining models

We need the `Author`, `Category`, and `Review` models. Here is how we can do this.

The `app/models/author.rb` file contains the following code:

```
class Author  
  include Mongoid::Document  
  
  field :name, type: String  
  
  has_many :books  
end
```

The `app/models/category.rb` file contains the following code:

```
class Category  
  include Mongoid::Document  
  
  field :name, type: String  
  
  has_and_belongs_to_many :books  
end
```

Note that the category and books have a many-to-many relationship. The `app/models/review.rb` file contains the following code:

```
class Review  
  include Mongoid::Document  
  
  field :comment, type: String  
  field :username, type: String  
  
  embedded_in :book  
end
```

It's very important that the inverse relation that is, the `embedded_in` is mentioned in reviews. This tells Mongoid how to store the embedded object. If this is not written, objects will not be embedded.

Testing from the Rails console

Nothing is ever complete without testing. The Rails community is almost fanatical about integrating tests into the project. We shall learn about testing soon, but for now let's test our code from the Rails console.

Time for action – putting it all together

Now we shall test these models to see if they indeed work as expected. We shall create different objects and their relations. The fun begins! Let's start the Rails console and create our first book object:

```
$ rails console
```



The Rails console is a command-line interactive command prompt that loads the Rails environment and the models. It's the best way to check and test if our data models are correct.

Let's create a book now. We can do that using the following code:

```
> b = Book.new(title: "Oliver Twist", publisher: "Dover Publications",  
published_on: Date.parse("2002-12-30") )
```

```
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: "Oliver  
Twist", publisher: "Dover Publications", published_on: 2002-12-30  
0:00:00 UTC, votes: nil, author_id: nil, category_ids: [] >
```

Here, we have populated the basic `title`, `publisher`, and `published_on` fields. Now let's work with the relations. Let's create an author, which can be done as follows:

```
> Author.create(name: "Charles Dickens")
```

```
=> #<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name: "Charles  
Dickens" >
```

Let's create a couple of categories too. This can be done as follows:

```
> Category.create(name: "Fiction")
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:
"Fiction", book_ids: []>

> Category.create(name: "Drama")
=> #<Category _id: 4e86e4d9fed0eb0be0000013, _type: nil, name: "Drama",
book_ids: []>
```

Now, let's add an author and some categories to our book. This can be done as follows:

```
> b.author = Author.where(name: "Charles Dickens").first
=> #<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name: "Charles
Dickens">

> b.categories << Category.first
=> []

> b.categories << Category.last
=> []

> b
=> #<Book _id: 4e86df21fed0eb0be000000b, _type: nil, title: "Oliver
Twist", publisher: "Dover Publications", published_on: 2002-12-30
00:00:00 UTC, votes: nil, author_id: BSON::ObjectId('4e86e4b6fed0eb0
be0000011'), category_ids: [BSON::ObjectId('4e86e4cbfed0eb0be0000012'),
BSON::ObjectId('4e86e4d9fed0eb0be0000013')]>

> b.save
=> true
```

Remember to save the object!



Save returns true if the object was saved successfully, otherwise it returns false. Save will raise an exception if the save was unsuccessful.

What just happened?

We have just created books, authors, and categories.

Hmm... category and books have a many-to-many relationship. So does this mean that category objects should also be updated? Let's check:

```
> Category.first  
  
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:  
"Fiction", book_ids: [BSON::ObjectId('4e86e45efed0eb0be0000010')]>  
> Category.last  
  
=> #<Category _id: 4e86e4d9fed0eb0be0000013, _type: nil, name: "Drama",  
book_ids: [BSON::ObjectId('4e86e45efed0eb0be0000010')]>
```

Yeah!, we are in good shape.

Let's check what MongoDB has stored. Start the Mongo CLI and see the books.

We can do this as follows:

```
$ mongo  
MongoDB shell version: 1.8.3  
connecting to: test  
  
> use sodibee_development  
switched to db sodibee_development  
  
> db.books.findOne()  
{  
  "_id" : ObjectId("4e86e45efed0eb0be0000010"),  
  "category_ids" : [  
    ObjectId("4e86e4cbfed0eb0be0000012"),  
    ObjectId("4e86e4d9fed0eb0be0000013")  
  ],  
  "name" : "Oliver Twist",  
  "publisher" : "Dover Publications",  
  "published_on" : ISODate("2002-12-30T00:00:00Z"),  
  "author_id" : ObjectId("4e86e4b6fed0eb0be0000011")  
}  
>
```

And let's see the categories and author objects too

```
> db.categories.findOne()
{
    "_id" : ObjectId("4e86e4cbfed0eb0be0000012"),
    "book_ids" : [
        ObjectId("4e86e45efed0eb0be0000010")
    ],
    "name" : "Fiction"
}

> db.categories.findOne({name: "Drama"})
{
    "_id" : ObjectId("4e86e4d9fed0eb0be0000013"),
    "book_ids" : [
        ObjectId("4e86e45efed0eb0be0000010")
    ],
    "name" : "Drama"
}

> db.authors.findOne()
{ "_id" : ObjectId("4e86e4b6fed0eb0be0000011"), "name" : "Charles Dickens" }
```

All is well!

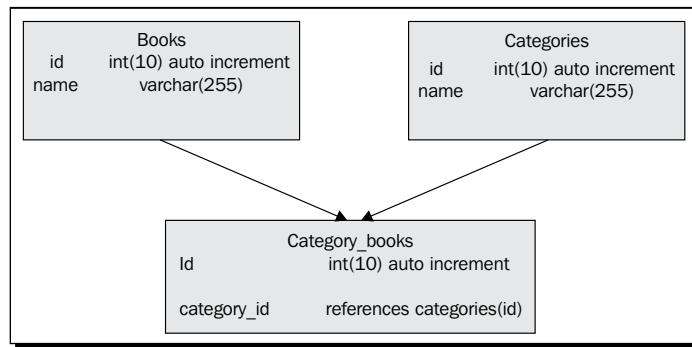
Have a go hero – adding more books, authors, and categories

Let's get creative (and funny) by adding the following:

- ◆ Adventures of Banana Man by Willie Slip in the Adventure category.
- ◆ World's craziest Moments and Dizzying moments by Mary Go Round in the Travel category.
- ◆ Procrastinate and Laziness Personified by Toby D Cided in the Self-help category

Understanding many-to-many relationships in MongoDB

In a SQL database, a many-to-many relationship is done using an intermediate table. For example, the many-to many relationship we have mentioned previously between books and categories, would be achieved in the following manner in a SQL database:



As MongoDB is a schemaless database, we do not need any additional temporary collections. The following is what the book object stores:

```
> db.books.findOne()
{
    "_id" : ObjectId("4e86e45efed0eb0be0000010"),
    "category_ids" : [
        ObjectId("4e86e4cbfed0eb0be0000012"),
        ObjectId("4e86e4d9fed0eb0be0000013")
    ],
    "name" : "Oliver Twist",
    "publisher" : "Dover Publications",
    "published_on" : ISODate("2002-12-30T00:00:00Z"),
    "author_id" : ObjectId("4e86e4b6fed0eb0be0000011")
}
>
```

The following is what the category object stores:

```
> db.categories.findOne()
{
    "_id" : ObjectId("4e86e4cbfed0eb0be0000012"),
    "book_ids" : [
        ObjectId("4e86e45efed0eb0be0000010")
    ]
}
```

```
ObjectId("4e86e45efed0eb0be0000010")
],
"name" : "Fiction"
}
```

No intermediate collections needed!

Using embedded documents

When we built the models, we embedded reviews in the book mode. An example would be ideal to explain this.

Time for action – adding reviews to books

Let's start the Rails console again and add reviews to books. This is done as follows:

```
> b = Book.where(title: "Oliver Twist").first

=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: "Oliver
Twist", publisher: "Dover Publications", published_on: 2002-12-30
00:00:00 UTC, votes: nil, author_id: nil, category_ids: []>

> b.reviews.create(comment: "Fast paced book!", username: "Gautam")

=> #<Review _id: 4e86f6c8fed0eb0be0000019, _type: nil, comment: "Fast
paced book!", username: "Gautam">

> b.reviews.create(comment: "Excellent literature", username: "Tom")

=> #<Review _id: 4e86f6ffffed0eb0be000001a, _type: nil, comment:
"Excellent literature", username: "Tom">
```

What just happened?

That's it—we just created reviews for books. Let's fetch them and check:

```
b.reviews
```

```
=> [#<Review _id: 4e86f68bfed0eb0be0000018, _type: nil,
comment: "Fast paced book!", username: "Gautam">, #<Review _id:
4e86f6ffffed0eb0be000001a, _type: nil, comment: "Excellent literature",
username: "Tom">]
```

Let's look at the following code to see what was stored in MongoDB:

```
> db.books.findOne()
{
    "_id" : ObjectId("4e86e45efed0eb0be0000010"),
    "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),
    "category_ids" : [
        ObjectId("4e86e4cbfed0eb0be0000012"),
        ObjectId("4e86e4d9fed0eb0be0000013")
    ],
    "name" : "Oliver Twist",
    "published_on" : ISODate("2002-12-30T00:00:00Z"),
    "publisher" : "Dover Publications",
    "reviews" : [
        {
            "comment" : "Fast paced book!",
            "username" : "Gautam",
            "_id" : ObjectId("4e86f68bfed0eb0be0000018")
        },
        {
            "comment" : "Excellent literature",
            "username" : "Tom",
            "_id" : ObjectId("4e86f6ffffed0eb0be000001a")
        }
    ]
}
>
```

Notice that the reviews are embedded inside the book object. Now when we fetch the book object, we will automatically get all the reviews too.

Choosing whether to embed or not to embed

Suppose we want to prepare orders for a book. The book can be leased or purchased. If we want to maintain an order history in terms of lease and purchase, how do we build the Lease, Purchase, and Order models?

Time for action – embedding Lease and Purchase models

We have three model files Order, Lease, and Purchase as follows:

```
# app/models/order
class Order
  include Mongoid::Document

  field :created_at, type: DateTime
  field :type, type: String # Lease, Purchase

  belongs_to :book

  embeds_one :lease
  embeds_one :purchase

end
```

Now, depending on the type field, we can determine which embedded object to pick up, the lease, or the purchase. You can design the Lease and Purchase models as shown in the following code:

```
# app/models/lease.rb
class Lease
  include Mongoid::Document

  field :from, type: DateTime
  field :till, type: DateTime

  embedded_in :order
end

# app/models/purchase.rb
class Purchase
  include Mongoid::Document

  field :quantity, type: Integer
  field :price, type: Float

  embedded_in :order
end
```

Working with Map/Reduce

To see an example of how Map/Reduce works, let's now add votes to books. The following shows how we can add votes:

```
{  
    "username" : "Dick",  
    "rating" : 5  
}
```

Rating could be on a scale of 1 to 10, with 10 being the best. Every user can rate a book. Our aim is to collect the total rating by all users. We shall save this information as a hash in the votes array in the book object. This should not be confused with an embedded object (as it does not have an object ID).



We have not seen the MongoDB data types such as ObjectId and ISODate. We shall learn about these data types in the future chapters. All usual data types such as integer, float, string, hash, and array are supported.

The following is how we save this information as a hash in the votes array in the book object:

```
> db.books.findOne()  
{  
    "_id" : ObjectId("4e86e45efed0eb0be0000010"),  
    "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),  
    "category_ids" : [  
        ObjectId("4e86e4cbfed0eb0be0000012"),  
        ObjectId("4e86e4d9fed0eb0be0000013")  
    ],  
    "name" : "Oliver Twist",  
    "published_on" : ISODate("2002-12-30T00:00:00Z"),  
    "publisher" : "Dover Publications",  
    "reviews" : [  
        {  
            "comment" : "Fast paced book!",  
            "username" : "Gautam",  
            "_id" : ObjectId("4e86f68bfed0eb0be0000018")  
        },  
        {  
            "comment" : "Excellent literature",  
            "username" : "Tom",  
            "_id" : ObjectId("4e86f6ffffed0eb0be000001a")  
        }  
    ],  
}
```

```
"votes" : [
    {
        "username" : "Gautam",
        "rating" : 3
    }
]
```

Before we see the example of Map/Reduce, it would be fun to add more books and votes, so that the Map/Reduce results make more sense. This is done as shown next:

```
> Book.create(name: "Great Expectations", author: Author.first)

=> #<Book _id: 4e8704fdfed0eb0f97000001, _type: nil, title: nil,
publisher: nil, published_on: nil, votes: nil, author_id: BSON::Object
objectId('4e86e4b6fed0eb0be0000011'), category_ids: [], name: "Great
Expectations">

> Book.create(name: "A tale of two cities", author: Author.first)

=> #<Book _id: 4e870521fed0eb0f97000002, _type: nil, title: nil,
publisher: nil, published_on: nil, votes: nil, author_id: BSON::Object
Id('4e86e4b6fed0eb0be0000011'), category_ids: [], name: "A tale of two
cities">
```

Now let's add votes for all three books.

First, for Oliver Twist (for example, one vote by Gautam)

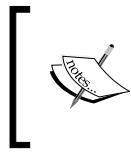
```
a = Book.first

=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", published_on: 2002-12-30 00:00:00 UTC,
votes: nil, author_id: BSON::ObjectId('4e86e4b6fed0eb0be0000011'),
category_ids: [BSON::ObjectId('4e86e4cbfed0eb0be0000012'), BSON::ObjectId
('4e86e4d9fed0eb0be0000013')], name: "Oliver Twist">

> b.votes = []
=> []

> b.votes << {username: "Gautam", rating: 3} => [{:username=>"Gautam",
:rating=>3}]

> b.save
=> true
```



Note that we first set `b.votes = []`, that is, an empty array. This is because MongoDB does not add the fields to the database until they are populated. So, by default `b.votes` would return nil. Hence it's important to initialize it the first time.

Now, for Great Expectations (for example, three votes, one each by Gautam, Tom, and Dick)

```
> b = Book.where(name: "Great Expectations").first  
  
=> #<Book _id: 4e8704fdfed0eb0f97000001, _type: nil, title: nil,  
publisher: nil, published_on: nil, votes: nil, author_id: BSON::Ob  
jectId('4e86e4b6fed0eb0be0000011'), category_ids: [], name: "Great  
Expectations">  
  
> b.votes = []  
=> []  
  
> b.votes << {username: "Gautam", rating: 9}  
=> [{:username=>"Gautam", :rating=>9}]  
  
> b.votes << {username: "Tom", rating: 3}  
=> [{:username=>"Gautam", :rating=>9}, {:username=>"Tom", :rating=>3}]  
  
> b.votes << {username: "Dick", rating: 7}  
=> [{:username=>"Gautam", :rating=>9}, {:username=>"Tom", :rating=>3},  
{:username=>"Dick", :rating=>7}]  
  
> b.save  
=> true
```

Finally, for The Tale of Two Cities (for example, two votes, one each by Gautam and Dick)

```
> c = Book.where(name: /cities/).first  
  
=> #<Book _id: 4e870521fed0eb0f97000002, _type: nil, title: nil,  
publisher: nil, published_on: nil, votes: nil, author_id: BSON::Object  
Id('4e86e4b6fed0eb0be0000011'), category_ids: [], name: "A tale of two  
cities">
```

```

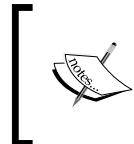
> c.votes = []
=> []
> c.votes << {username: "Gautam", rating: 9}
=> [{:username=>"Gautam", :rating=>9}]

> c.votes << {username: "Dick", rating: 5}
=> [{:username=>"Gautam", :rating=>9}, {:username=>"Dick", :rating=>5}]

> c.save
=> true

```

If we want to collect all the votes and add up the rating for each user, it can be a pretty cumbersome task to iterate over all of these objects. This is where Map/Reduce helps us.



One alternative to Map/Reduce in this particular example would be to capture the vote count per book by incrementing a counter while inserting votes and reviews itself. However, we shall use Map/Reduce here so that we understand how it works.



Time for action – writing the map function to calculate ratings

This is how we can write the map function. As we have seen earlier, this function will emit information, in our case, the key is the username and the value is the rating:

```

function() {
    this.votes.forEach(function(x) {
        emit(x.username, {rating: x.rating});
    });
}

```

What just happened?

This is a JavaScript function. MongoDB understands and processes all JS functions. Every time `emit()` is called, some data is emitted for the reduce function to process. In the preceding code `this` represents the collection object.

What we want to do is emit all the ratings for each element in the `votes` array for every book. The `emit()` takes the key and value as parameters. So, we are emitting the users votes for the reduce function to process. It's also important to remember the data structure we are emitting as the value. It should be consistent for all objects. In our case `{rating: x.rating}`.

Time for action – writing the reduce function to process the emitted results

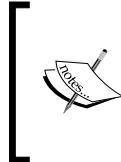
Now let's write the reduce function. This takes a key and an array of values, shown as follows:

```
function(key, values) {  
    var result = {rating: 0};  
  
    values.forEach(function(value) {  
        result.rating += value.rating;  
    });  
  
    return result;  
}
```

What just happened?

The reduce function is the one which processes the values that were emitted from the map function.

Remember that the `values` parameter is always an array. The map function could emit results for the same key multiple times, so we should be sure to process the value as an array and accumulate results. The return structure should be the same as what was emitted.



MongoDB supports Map/Reduce and will invoke Map/Reduce functions in parallel. This gives it power over standard SQL databases. The closest a SQL database comes to this is when we use a GROUP BY query. It depends on the indexes and the query fired that can get us similar results like Map/Reduce.

Using Map/Reduce together

As MongoDB requires JavaScript functions, the trick here is to pass the JavaScript functions to the MongoDB engine via a string on the Rails console. So, we create two strings for the map and reduce functions.

Time for action – working with Map/Reduce using Ruby

We shall now create two strings in Ruby for these functions:

```
> map = %q{function() {
    this.votes.forEach(function(x) {
        emit(x.username, {rating: x.rating});
    });
}
}

> reduce = %q{function(key, values) {
    var result = {rating: 0};
    values.forEach(function(value) {
        result.rating += value.rating;
    });
    return result;
}
}
```



%q is an efficient, clean, and optimized way of writing multiline strings in Ruby!

Remember that we are now in the MongoDB realm, so we should not work on Ruby objects but only on the MongoDB collection. So, we call `map_reduce` on the `book` collection, as follows:

```
> results = Book.collection.map_reduce(map, reduce, out: "vr")

=> #<Mongo::Collection:0x20cf7a4 @name="vr", @db=#<Mongo::DB:0x1ab8564 @
name="sodibee_development",
...
...
@cache_time=300, @cache={}, @safe=false, @pk_factory=BSON::ObjectId, @
hint=nil>
```

The output you saw previously is the MongoDB collection Map/Reduce result. Let's fetch the full results now. The following command does it for us:

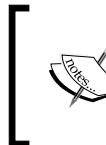
```
> results.find().to_a  
  
=> [{"_id"=>"Dick", "value"=>{"rating"=>12.0}}, {"_id"=>"Gautam",  
"value"=>{"rating"=>21.0}}, {"_id"=>"Tom", "value"=>{"rating"=>3.0}}]
```

What just happened?

Voila! This shows that we have the following result:

- ◆ Dick has 12 ratings
- ◆ Gautam has 21 ratings
- ◆ Tom has 3 ratings

Tally these ratings manually with the preceding code and verify.



What would you have to do if you did not have Map/Reduce?
Iterate over all book objects and collect the votes array. Then
keep a temporary hash of usernames and keep aggregating the
ratings. Lots of work indeed!

Don't always jump into using Map/Reduce. Sometimes it's just easier to query properly.
Suppose, we want to find all the books that have votes or reviews for them, what do we do?

- ◆ Do we iterate every book object and check the length of the votes array or the reviews array?
- ◆ Do we run Map/Reduce for this?
- ◆ Is there a direct query for this?

We can directly fire a query from the Rails console, as follows:

```
irb> Book.any_of({:reviews.exists => true}, {:votes.exists => true})
```

If we want to search directly on the mongo console, we have to execute the following command:

```
mongo> db.books.find({"$or": [{reviews: {"$exists" : true}}, {votes : {"$exists": true}}]})
```

Remember, we should use Map/Reduce only when we have to process data and return results (for example, when it's mostly statistical data). For most cases, there would be a query (or multiple queries) that would get us our results.

Pop quiz – swimming in MongoDB and Ruby

1. How does MongoDB store data?
 - a. As JSON.
 - b. As Binary JSON or BSON.
 - c. As text in files.
 - d. An encrypted binary file.
2. What are collections in MongoDB?
 - a. Collections store documents.
 - b. Collections store other collections.
 - c. There is no such thing as collections.
3. How do we represent an array of hashes in MongoDB?
 - a. Arrays can only have strings or integers in them.
 - b. Like this [{ k1: "v1" }, { k1: "v2" }].
 - c. Hashes are not supported in MongoDB.
 - d. Like this { k1: ["v1", "v2"], k2: ["v1", "v2"] }.
4. Which answer represents one of the ways models in Ruby communicate with MongoDB?
 - a. Models in Ruby cannot talk directly to MongoDB.
 - b. Install the BSON gem.
 - c. Install the Mongoid gem and include Mongoid::Document in the Ruby class.
 - d. We inherit the Ruby class from ActiveRecord::Base.
5. How are many-to-many relationships mapped in MongoDB?
 - a. We create a third collection to store ObjectId instances.
 - b. Many-to-many is not supported in MongoDB.
 - c. Each document saves the other in an Array field inside it.
 - d. Only one document saves information about the other.

6. How can we create a join of two collections in MongoDB?
 - a. We cannot! Joins are not supported in MongoDB.
 - b. db.collection1.find({ \$join: "collection2" }).
 - c. Always use Map/Reduce instead of joins.
 - d. db.join({ collection1: 1, collection2: 1 }).

Summary

Here we really jumped into Ruby and MongoDB, didn't we? We saw how to create objects in MongoDB directly and then via Ruby using Mongoid. We saw how to set up a Rails project, configure Mongoid, and build models. We even went the distance to see how Map/Reduce would work in MongoDB.

We saw a lot of new things too, which require explanation. For example, the various data types that are supported in MongoDB, such as ObjectId, ISODate.

In the next chapter, we shall dive deeper in these internal concepts and understand more about how MongoDB works. Hang on tightly!

3

MongoDB Internals

Now that we have had a brief look at Ruby and MongoDB interactions via Mongoid, I believe it is the right time to know what happens under the hood. This information is good to know but not mandatory. If you are a person in the fast lane, you can skip this chapter and go straight to Chapter 4, Working Out Your Way with Queries.

In this chapter we shall learn:

- ◆ What exactly MongoDB documents and objects are.
- ◆ What is BSON and how is it used in MongoDB to save information?
- ◆ How and why does MongoDB use JavaScript?
- ◆ What are MongoDB journal entries; how and why are they written?
- ◆ What is the global write lock and how does it function?
- ◆ Why are there no joins in MongoDB?

We have seen some examples of MongoDB objects earlier; these objects look similar to JSON objects. However, MongoDB does not use JSON to store information – it uses **Binary JSON (BSON)** for storage. Using BSON has a lot of advantages that we shall soon see.

Understanding Binary JSON

The following is a sample of a JSON object we have seen before:

```
{  
    "_id" : ObjectId("4e86e45efed0eb0be0000010"),  
    "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),  
    "category_ids" : [  
        ObjectId("4e86e4cbfed0eb0be0000012"),  
        ObjectId("4e86e4d9fed0eb0be0000013")  
    ],  
    "name" : "Oliver Twist",  
    "published_on" : ISODate("2002-12-30T00:00:00Z"),  
    "publisher" : "Dover Publications"  
}
```

There is a strange JSON output here (that I refrained from explaining earlier) for `ObjectId` and `ISODate`. What is even stranger is that this data is not saved to the disk in the same format as shown in the preceding code. Instead it is saved as Binary JSON—a serialized JSON string. The following is a simple example:

```
{"hello": "world"}
```

Every BSON data has the following format:

```
<size> <type> <null byte>
```

The data in the preceding example is stored on the disk in the following format:

```
\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00
```

This is explained as follows:

- ◆ \x16\x00\x00\x00: This indicates that the size of the binary data is 22 bytes (remember 16 hex is 22 decimal)
- ◆ \x02: This indicates that the value is a BSON string
- ◆ hello\x00: The is the key that is always a null terminated string.
- ◆ \x00: The BSON value has been identified as a null terminated string.

You might ask, "Why not just plain old `{ "hello" : world }`?" There are plenty of reasons:

- ◆ Binary data is easier to store and manipulate
- ◆ Binary data is packed, so it consumes less space
- ◆ Insertions and deletions in binary embedded objects are easy

Of course, more explanations are due!

Fetching and traversing data

As the data is in BSON format, it's easy to traverse it. The first 4 bytes tell us how much data is stored, so that objects can be easily skipped without parsing the data. It's easy to skip embedded data too, as all the size of the data is known.

Manipulating data

When an embedded document is manipulated, MongoDB simply calculates the offset and reaches it. Now, when some data is changed or added to this embedded objects, we don't need to write the entire object back to the disk—MongoDB simply updates that BSON document and the length of the data. This is quick and clean.

What is ObjectId?

`ObjectId` is a unique ID for a document. It is a 12-byte binary value designed to have a reasonably high probability of being unique when allocated. By default the `ObjectId` field is stored under `_id`.

The concept of a unique Object ID as a primary key is important for MongoDB. In a highly scalable system, this ensures that an Object ID "almost" never repeats. The first 4 bytes of `ObjectId` indicate the time (in seconds) since epoch and the last 3 bytes represent a counter. Even if you insert two documents at the same moment, the counter value should increase.



There is nothing called guaranteed unique IDs—but it's almost guaranteed. According to Wikipedia, "Only after generating 1 billion UUIDs every second for the next 100 years, the probability of creating just one duplicate would be about 50%". Object IDs are not UUIDs but guarantee uniqueness.

Object ID is generated using the timestamp, 3 bytes of the MD5 hash of the machine name, its MAC address or a virtual machine ID, the process ID, and an ever incrementing value. Though every object has a unique ID, you would notice incrementing values for object IDs.

Documents and collections

Documents in MongoDB are structured documents saved in BSON format as mentioned in the earlier section. The maximum size of documents is 16 MB. It's interesting to note that 16 MB is not a limitation but is maintained for the sake of sanity!

In case we are required to store documents larger than 16 MB, MongoDB may be the wrong choice. For storing large documents, such as videos, GridFS is recommended.

Documents are analogous to records and are stored in collections, which are analogous to database tables. Documents in a collection are usually structured similarly but it's not mandatory. That means you can have differently structured documents in the same collection. That's the essence of NoSQL or a "schema-free" database.

Collections can be scoped or namespaces. For example, we could have a collection `rack` which has `shelves` and `panels` in it. These collections have other collections inside them:

```
db.rack
db.rack.shelves
db.rack.shelves.sections

db.rack.panels
db.rack.panels.components
```

Capped collections

Capped collections have a fixed number of documents in them. They can be considered as a "queue" that discards the oldest element when the cap is reached. The ideal example for this is log entries. We create capped collections as follows:

```
Db.createCollection("myqueue", {capped: true, size: 10000})
```

Dates in MongoDB

Dates are saved independent of the time zone. They are always stored as epoch time—the time in seconds from January 1, 1970.

```
> new ISODate("2011-12-31T12:01:02+04:30")
ISODate("2011-12-31T07:31:02Z")

> new ISODate("sdf")
Tue Nov  8 08:14:49 uncaught exception: invalid ISO date

> new ISODate("garbage 2011-12-31T12:01:02+05:30 more garbage")
ISODate("2011-12-31T06:31:02Z")
```

JavaScript and MongoDB

JavaScript seems a strange choice for a database for server-side code execution. However, it's definitely a better choice than writing a custom language syntax—JavaScript is a very popular language, well known among developers, and just like MongoDB it's evolving fast too.

We have already seen the use of JavaScript in Map/Reduce functions. But we can do more than that. We can write our own custom JavaScript functions and call them when we want. Consider them more like stored procedures written in JavaScript.

`db.eval` is a function that is used to evaluate custom JavaScript functions that we write.

Time for action – writing our own custom functions in MongoDB

Let's say we want to write a function to delete authors that don't have any books, we can write this in JavaScript as follows:

```
function rid_fakes() {
    var ids = [];
    db.authors.find().forEach( function(obj) {
        if (db.books.find({author_id: obj._id}).length() == 0 ) {
            ids.push(obj._id);
        }
    });
    db.authors.remove({_id : { $in : ids }});
}

db.eval(rid_fakes);
```



In a Ruby app, it's recommended to manage the objects rather than the documents. This is to ensure that the cache does not get corrupted.



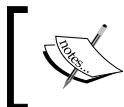
Ensuring write consistency or "read your writes"

It's very important to ensure that the database is eventually consistent. As we shall soon see, MongoDB delays all writes to the disk because the disk's I/O is slow. Write consistency means that every time something is written to the database, the delayed write should not cause inconsistency when we read back the data. MongoDB ensures this consistency for every write operation and the updated value is always returned back in the read operation. This is important for a couple of reasons:

- ◆ Ensuring you always get the latest updated data
- ◆ Easy and consistent crash recovery

How does MongoDB use its memory-mapped storage engine?

MongoDB tries to be as efficient and fast as it can get. So, to cater to this, it uses memory-mapped files for storage. This is as fast as it can get with the disk I/O and system cache. As every operating system works with virtual memory, MongoDB leverages this and can effectively be as large as the virtual memory allows it to be.



Memory-mapped files are segments of virtual memory that are mapped byte-for-byte between the file and the memory. So, they can be considered as fast as primary memory.



This also has an inherent advantage that as the operating system's virtual memory management gets better, it automatically improves the performance of the database storage engine too!

There is a downside to everything! Memory-mapped files store information in the memory and sync to the database after a short while (by default in MongoDB that is 100 ms). So, we are indeed dealing with a database where we could potentially lose the last 100 ms of information.

Advantages of write-ahead journaling

MongoDB (v1.7.5 onwards) supports **write-ahead journaling**. This means that before the data is written to the collections, it is written to the journal. This ensures that there is always write consistency. For every write to the database:

1. Information is first written to the journal.
2. After the journal entry is synchronized to the disk, data is written to the database's memory-mapped file.
3. Information is then synchronized to the disk.

It's important to know that when a MongoDB client writes to the database, it is guaranteed to return the updated result. If journaling fails, the entire write operation is deemed failed. Journaling can be turned off but it's strongly recommended to be enabled.

Global write lock

I mentioned earlier that MongoDB writes to the disk (using `fsync`) every 100 ms. However, when this data is being written to the disk, it's important to keep it consistent. Hence, MongoDB, for quite some versions, used a global write lock to ensure this.

This creates a problem because the entire database is locked until the write is complete. This means that if we have a long running write query, the database is locked for good and the performance and efficiency is seriously hit.

The later versions of MongoDB (at the time of writing) plan to implement a collection-based lock to ensure that we can write simultaneously across collections – but it's not there today.

What it does have instead is lock yielding. That means, any MongoDB thread will yield their lock on page faults or long running queries. This solves the problem of the global lock to a level of acceptable efficiency. This is also called **interleaving**—when a long running write is in progress, the thread yields temporarily for intermediate reads and writes.

Transactional support in MongoDB

MongoDB's primary objectives are to manage large data, be fast, and scale easily! So, it's never going to be a perfect fit for all applications. This has been the source of debate between the SQL and NoSQL factions.

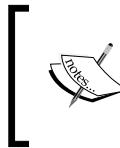
From a practical perspective, we should know there are no ACID transactions in MongoDB. There are a few ways to do transactions in MongoDB but it may not always be a suitable choice. Basically if you require a multi-document transaction, such as financial data that is spread across different collections, MongoDB may be the wrong choice. However, for most web applications, transactional support is usually a sanity check and not a complex rollback. In any case, choose wisely!

Understanding embedded documents and atomic updates

All document updates in MongoDB are atomic. This can itself be a very easy way to simulate transactional support in MongoDB. For example, if we require `Orders` to be created with `LineItems`, we can easily simulate a transaction by embedding `LineItems` into `Order`. That way when the document is saved, we are guaranteed atomic transactions.

Implementing optimistic locking in MongoDB

We can do optimistic locking using lock versioning. First let's understand what this means. Every time the document, object, record, or row in the database is updated, we increment a value of the field. When we read the document, we know the value of the field. When we want to save the document, we ensure that the value we had read earlier has not changed. If it's different, it means someone updated the document before us—so we need to read it again. This is also called **Compare and Set (CAS)**.



Optimistic locking already exists in ActiveRecord. If you simply add a column called `lock_version` in your table, it starts optimistic locking. `StateObjectError` is raised in case the document's `lock_version` value has changed.

Time for action – implementing optimistic locking

Let's add a field in our document called `lock_version` and set its initial value as 0.

When we fetch this object, we know what the version is. So, when we fire the update call, we ensure that it's part of the object selector!

```
mongo> db.authors.findOne()
{
  "_id" : ObjectId("4f81832efed0eb0bbb000002"),
  "name" : "Victor Metz",
  "_type" : "Author",
  "lock_version" : 0
}

mongo> db.authors.update({ _id: ObjectId("4f81832efed0eb0bbb000002"),
  lock_version: 0 }, {name: "Victor Matz", lock_version: 1})

mongo> db.authors.find({ _id: ObjectId("4f81832efed0eb0bbb000002") })
{ "_id" : ObjectId("4f81832efed0eb0bbb000002"), "name" : "Victor
Metz", "_type" : "Author", "lock_version" : 1 }

mongo> db.authors.update(db.authors.update({ _id: ObjectId("4f81832ef
ed0eb0bbb000002"), lock_version: 0 }, {name: "NO SUCH AUTHOR", lock_
version: 1}))

mongo> db.authors.find({ _id: ObjectId("4f81832efed0eb0bbb000002") })
{ "_id" : ObjectId("4f81832efed0eb0bbb000002"), "name" : "Victor
Metz", "_type" : "Author", "lock_version" : 1 }
```

What just happened?

What's important is to keep a check on the `lock_version` field. When we fetched the first author objects, the `lock_version` value was 0.

```
mongo> db.authors.update(
  { _id: ObjectId("4f81832efed0eb0bbb000002"), lock_version: 0 },
  {name: "Victor Matz", lock_version: 1})
```

We are not just updating an object that has an ID equal to `4f81832efed0eb0bbb000002` but also where the `lock_version` field is set. Notice that `lock_version` is being updated. This is a programmer's instruction. If we don't update `lock_version` manually, this strategy would fail! Now we have `lock_version` set at value 1. If we tried to update the object as shown in the following code snippet, the object selection would fail and the object would not be updated:

```
mongo> db.authors.update(
  { _id: ObjectId("4f81832efed0eb0bbb000002") }, { lock_version: 0 },
  { name: "NO SUCH AUTHOR", lock_version: 1 })
```

If that object has been modified by some other process or thread, `lock_version` would have been incremented. So, the object in our preceding query would not get updated if the lock version changes. But how do we do this in our Ruby program?



How do we perform Optimistic locking using Mongoid?

There are a few extensions available for this. See an example here at https://github.com/burgalon/mongoid_optimistic_locking. Basically, this changes the `atomic_selector` method to include a `_lock_version` field and auto-increment it on every save!

Choosing between ACID transactions and MongoDB transactions

Finally, we have seen how we can manipulate data safely using atomic operations and ensure data consistency. However, where you require transactions that span multiple documents or tables and that is a critical feature of your application, consider not using MongoDB.

For everything else, there's MongoDB.

Why are there no joins in MongoDB?

Joins are good, they say! And for a good reason, normalization is the best option! Let's say we have `authors`, `books`, and `orders`. What if we wanted to find the orders of books sold by authors that have the name `Mark`? An SQL query would probably be something like the following query:

```
SELECT * FROM orders, books, authors WHERE books.author_id = author.id
AND orders.book_id = book.id AND author.first_name LIKE "Mark%"
```

This causes an implicit join between `authors`, `books`, and `orders`. This is fine only under the following circumstances:

- ◆ The data in `authors`, `books`, and `orders` is not huge! If we had 1 million entries in each table, it could reach a temporary join of around 1 million * 1 million * 1 million entries, degrading the performance drastically. Every RDBMS is smart enough not to create such a huge temporary table of course, but the result set is still huge.
- ◆ If we consider that the data is distributed between nodes (shared), the network latency to gather information for a join from different nodes is going to be huge.

These are a few reasons why the NoSQL faction shies away from joins. As we have seen earlier, the priorities for MongoDB is managing huge data with easy scaling, sharing, and faster querying. So, what are the alternatives to joins? Plenty!

- ◆ The simplest solution is to fire multiple queries and programmatically get your results set. As querying is fast, the cumulative time taken by firing multiple queries could be compared to a fancy single query join, if not faster!
- ◆ Denormalize and duplicate data—sometimes, it's just easier to add some redundant information if it's going to make querying faster.
- ◆ Use Map/Reduce techniques to distribute and gather data from the database.

Pop quiz – the dos and don'ts of MongoDB

1. Why does MongoDB use BSON and not just JSON?
 - a. MongoDB wants to be different!
 - b. BSON enables faster inline data manipulation and traversal.
 - c. BSON and JSON are the same.
 - d. MongoDB uses JSON and not BSON.
2. How does MongoDB persist data?
 - a. In memory-mapped files that are flushed to the disk every 100 ms.
 - b. Data is saved in the memory.
 - c. Data is saved in files on the disk.
 - d. Data is not saved.

3. Which of the following is true for MongoDB?
 - a. Joins and transactions are fully supported in MongoDB.
 - b. Joins are supported but transactions are not supported.
 - c. Joins and multi-collection transactions are not supported.
 - d. Single collection transactions are not supported.
4. What is write-ahead journaling in MongoDB?
 - a. Writes are written with a timestamp in the future.
 - b. Writes are written to the journal log first and then lazily to the disk.
 - c. Writes are written to the disk first and then to the journal log.
 - d. Writes are written only in the journal.

Summary

MongoDB has a lot of things going on under the covers, most of which we may either take for granted or sometimes do not need to know to work with MongoDB. The team behind MongoDB has been working hard to make MongoDB faster, easier, and more humongous. If we understand how things work and what impact it's going to have on our data or performance, it would help us build better applications by making the most of all that is offered by MongoDB. MongoDB does not support joins and transactions. There are alternatives to this but if you require ACID transactions, you should use an SQL database.

In the subsequent chapters, we shall learn a lot about using MongoDB but we may not see many MongoDB internals. I do hope that this chapter makes the underlying concepts easy to understand.

4

Working Out Your Way with Queries

Wherever there is a database, there has to be some search criteria! This chapter takes our journey forward towards searching for data in MongoDB. In this chapter we will see how we can search via the mongo console.

In this chapter we shall learn the techniques for:

- ◆ Searching by field attributes (such as strings, numbers, float, and date)
- ◆ Searching on indexed fields
- ◆ Searching by values inside an array field
- ◆ Searching by values inside a hash field
- ◆ Searching inside embedded objects
- ◆ Searching by regular expressions

Let's start searching with the help from our good old Sodibee database!

Searching by fields in a document

Let's consider a book structure like the following:

```
{  
  "_id" : ObjectId("4e86e45efed0eb0be0000010") ,  
  "author_id" : ObjectId("4e86e4b6fed0eb0be0000011") ,  
  "category_ids" : [  
    ObjectId("4e86e4cbfed0eb0be0000012") ,  
    ObjectId("4e86e4d9fed0eb0be0000013")  
  ] ,
```

```
"name" : "Oliver Twist",
"published_on" : ISODate("2002-12-30T00:00:00Z"),
"publisher" : "Dover Publications",
"reviews" : [
  {
    "comment" : "Fast paced book!",
    "username" : "Gautam",
    "_id" : ObjectId("4e86f68bfed0eb0be0000018")
  },
  {
    "comment" : "Excellent literature",
    "username" : "Tom",
    "_id" : ObjectId("4e86f6ffffed0eb0be000001a")
  }
],
"votes" : [
  {
    "username" : "Gautam",
    "rating" : 3
  }
]
```

We have already done this earlier, but let's reiterate and dig deeper. Let's find all the books published by Dover Publications. First let's start the mongo console as follows:

```
$ mongo
MongoDB shell version: 2.0.2
connecting to: test
> use sodibee
switched to db sodibee
```

Time for action – searching by a string value

Let's find all the books that were published by Dover Publications. The following code shows us how to accomplish this:

```
> db.find({ publisher : "Dover Publications"})

{
  "_id" : ObjectId("4e86e45efed0eb0be0000010"),
  "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),
  "category_ids" : [
    ObjectId("4e86e4cbfed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
```

```
[{"name": "Oliver Twist", "publisher": "Dover Publications", "reviews": [{"comment": "Fast paced book!", "username": "Gautam", "_id": ObjectId("4e86f68bfed0eb0be0000018")}, {"comment": "Excellent literature", "username": "Tom", "_id": ObjectId("4e86f6ffffed0eb0be000001a")}], "votes": [{"username": "Gautam", "rating": 3}]}]
```

What just happened?

We have just fired a simple `find()` query on a collection to help us get the relevant documents from the database. We can also configure the parameters in `find()` to get more specific details. To see what specific parameters `find()` has, issue the following command:

```
> db.books.find

function (query, fields, limit, skip) {
    return new DBQuery(this._mongo, this._db, this, this._fullName,
this._massageObject(query), fields, limit, skip);
}
```

The configuration parameters for `find()` in the preceding code are explained as follows:

- ◆ `query`: This is the selection criteria. For example, `{ publisher: "Dover Publications" }` as we had mentioned earlier. This is similar to the `WHERE` clause in a relational query.
- ◆ `fields`: These are the fields which we want selected. This is similar to the `SELECT` part of a query in a relational query. By default, all fields would be selected, so `SELECT *` is the default. In MongoDB we can specify inclusion as well as exclusion of fields. We will see an example of this shortly.
- ◆ `limit`: This represents the number of elements we want returned from the query. This is similar to the `LIMIT` part of a relational query.
- ◆ `skip`: This is the number of elements the query should skip before collecting results. This is similar to the `OFFSET` part of a relational query.

Have a go hero – search for books from an author

How do we search for books that are published by Dover Publications and written by Mark Twain?

Hint: We need to fire two queries. The first one would be to find the author by name "Mark Twain". Then using that `ObjectId`, we can find the books written by that author and published by Dover Publications.

Querying for specific fields

Let's now evaluate these options in greater detail.

Time for action – fetching only for specific fields

First, let's select only a few fields and see how the `fields` parameter works. This would be similar to an SQL query. For example:

```
SELECT name, published_on, publisher FROM books WHERE publisher =  
"Dover Publications";
```

In MongoDB this is achieved as follows:

```
> db.books.find({ publisher: "Dover Publications"}, {name: 1,  
published_on : 1, publisher : 1 })  
  
{ "_id" : ObjectId("4e86e45efed0eb0be0000010"), "name" : "Oliver  
Twist", "published_on" : ISODate("2002-12-30T00:00:00Z"), "publisher"  
: "Dover Publications" }
```

So far so good! But here is where MongoDB is more customizable and can do something that SQL cannot. Notice that the values for the selected fields are `1` (they can also be set to `true` instead of `1`). We can optionally set them to `0` or `false` and then these will be the fields excluded from the result. Let's see it in action in the following code:

```
> db.books.find({ publisher: "Dover Publications"}, {name: 0,  
published_on : 0, publisher : 0 })  
  
{ "_id" : ObjectId("4e86e45efed0eb0be0000010"),  
"author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),  
"category_ids" : [  
    ObjectId("4e86e4cbfed0eb0be0000012"),  
    ObjectId("4e86e4d9fed0eb0be0000013")  
, "reviews" : [  
    {
```

```

    "comment" : "Fast paced book!",
    "username" : "Gautam",
    "_id" : ObjectId("4e86f68bfed0eb0be0000018")
},
{
    "comment" : "Excellent literature",
    "username" : "Tom",
    "_id" : ObjectId("4e86f6ffffed0eb0be000001a")
}
], "votes" : [ { "username" : "Gautam", "rating" : 3 } ]
}

```

Notice that all fields are present in the result except `name`, `published_on`, and `publisher`.

What just happened?

Magic! Not only can we set inclusion fields but also exclusion fields. I don't believe there is any way to set exclusion fields in an SQL query.



Let me be fair here, SQL databases intentionally do not allow exclusion of fields from a `SELECT` query because of the structured nature of the tables, so as to ensure good performance and to ensure that the contract between the client-server is stable!

Imagine what happens to our query if we allow exclusion of columns and those columns are deleted—so many additional checks and degradation of performance! Code extremists would even say, you can fetch the data, filter it later, and remove the columns you don't want!

You can add more criteria to the query field and they will be set. This would be similar to the `AND` part in a `WHERE` clause.



Playing with inclusion and exclusion of fields

Remember that you cannot set inclusion and exclusion fields in the same query. This means either all the fields should have value 1 or all should have value 0. Otherwise MongoDB will throw an error **10053: You cannot currently mix including and excluding fields.**

The only exception to this is the exclusion of the `_id` field. We can exclude the `_id` field while including others. This means `db.books.findOne({}, {_id: 0, name: 1})` is valid.

Have a go hero – including and excluding fields

Well, go ahead and experiment with the following:

- ◆ Set different inclusion or exclusion fields for the books document.
- ◆ Set the limit and OFFSET for the query. Let me give you some hints here. A limit of 0 would mean no limit. skip values can be used for paging. Give it a shot and check a little later in the chapter whether you got it right!

Using skip and limit

skip and limit are both optional parameters to the find query. limit will limit the number of elements in the result and skip will skip elements in the result.

Time for action – skipping documents and limiting our search results

Suppose we want to query the second and third book in the collection. We can set the skip value to 1 or 2 and the limit value to 1. This is done as follows:

```
> db.books.find({}, {}, 1, 1)

{ "_id" : ObjectId("4e8704fdfed0eb0f97000001"), "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"), "category_ids" : [ ], "name" : "Great Expectations", "votes" : [
  {
    "username" : "Gautam",
    "rating" : 9
  },
  {
    "username" : "Tom",
    "rating" : 3
  },
  {
    "username" : "Dick",
    "rating" : 7
  }
] }

> db.books.find({}, {}, 1, 2)

{ "_id" : ObjectId("4e870521fed0eb0f97000002"), "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"), "category_ids" : [ ], "name" : "A tale of two cities", "votes" : [
```

```
{
  "username" : "Gautam",
  "rating" : 9
},
{
  "username" : "Dick",
  "rating" : 5
}
]
```

What just happened?

Notice that in both cases, we have mentioned the `query` and `fields` parameters as an empty hash. This is just for the sake of brevity!

`limit` is 1 in both cases but the `skip` values have changed. This would be similar to the following SQL query:

```
SELECT * FROM books LIMIT 1 OFFSET 1
```

Have a go hero – paginating document results

To see pagination in action, it would really be cool if you add 20 books to the collection. Then query them using the `limit` value as 10 with the `skip` value as 0 for getting results of page 1 and the `skip` value as 10 to get results of page 2.



There are utility methods such as `findOne()`, which just get us the first record. This has only two parameters: `query` and `fields`, as `skip` and `limit` would be irrelevant.



Writing conditional queries

We have seen how to query on multiple conditions. These were in conjunction, that is, they were bound by the `AND` clause:

```
> db.books.find({publisher: "Dover Publications", name: "Oliver Twist"})
```

This would be similar to an SQL query:

```
SELECT * FROM books WHERE publisher = "Dover Publications" AND name = "Oliver Twist";
```

Notice that `AND` is the default condition when multiple query parameters are specified. But this is not always the case!

Using the \$or operator

The `$or` operator is very common when we want a result set that satisfies any one of the conditions specified.

Time for action – finding books by name or publisher

Let's find all the books that have the name `Oliver Twist` or are from `Dover Publications`. For the sake of brevity, we shall select only the `name` field as follows:

```
db.books.find({ $or : [ { name: "Oliver Twist" } , { publisher : "Dover Publications" } ] })
```

This will give us our result set of books with either the name as `Oliver Twist` or publisher as `Dover Publications`.

What just happened?

The previous query is similar to the following:

```
SELECT * FROM books WHERE publisher = "Dover Publications" OR name = "Oliver Twist";
```

Let's look at the query parameters in a little more detail:

```
{$or : [
  {name: "Oliver Twist"},
  {publisher : "Dover Publications"}
]}
```

`$or` is a special operator in MongoDB and takes an array of query parameters. We can use this in conjunction with other parameters too:

```
db.books.find({ published_on: ISODate("2002-12-30") , $or : [ { name: "Oliver Twist" } , {publisher : "Dover Publications" } ] })
```

This would query with `AND` and `OR`. Its SQL equivalent would be:

```
SELECT * from books WHERE published_on = "2002-12-30" AND (name = "Oliver Twist" OR publisher = "Dover Publications");
```

Writing threshold queries with `$gt`, `$lt`, `$ne`, `$lte`, and `$gte`

We always require to search within a threshold, don't we?

MongoDB	SQL	Meaning
\$gt	>	Greater than
\$lt	<	Less than
\$gte	>=	Greater than or equal to
\$lte	<=	Less than or equal to
\$ne	!=	Not equal to

Time for action – finding the highly ranked books

Suppose we add the `rank` field to the books, our book object will look something as follows:

```
{
  "_id" : ObjectId("4e870521fed0eb0f97000002"),
  "rank" : 10
}
```

Now, if we want to search for all books having a rank in the top 10 ranks, we can fire the following query:

```
> db.books.find({ "rank" : { $lte : 10 } } )
```

You can add more operators in the same hash too. For example, if we want to find books in the top ten but not the top ranked book (that is, `rank != 1`), we can do the following:

```
> db.books.find({ "rank" : { $lte : 10, $ne : 1 } } )
```

Have a go hero – find books via rank

Why don't you give this a shot?

- ◆ Find books which have a rank between 5 and 10
- ◆ Find books before and after a particular date

Checking presence using \$exists

As MongoDB is schema free, there are times when we want to check the presence of some field in a document. For example, over the years, our schema for books evolved and we added some new fields. If we want to take a specific action on books that only have these new fields, we may need to check if these fields exist.

Suppose we want to search only for those books that have the `rank` field in them, it can be done as follows:

```
> db.books.find( { "rank" : { $exists : 1} } )
```

Searching inside arrays

Unlike most SQL databases, MongoDB can store values inside arrays and hashes. Now, we shall see how we can search inside arrays.



Did you know that most of the operators we learned about earlier, could be used directly on arrays inside a document just like normal fields? For example:

```
> db.books.insert( { "categories" : [ "Drama", "Action"] } )  
> db.books.find( { categories : { $ne : "Romance"} } )
```

This will return the document we inserted previously. Isn't that cool?!

Time for action – searching inside reviews

Let's now have a look at our `books` document. We have an array of reviews. A review is an embedded object (notice the `_id` parameter):

```
"reviews" : [  
    {  
        "comment" : "Fast paced book!",  
        "username" : "Gautam",  
        "_id" : ObjectId("4e86f68bfed0eb0be0000018")  
    },  
    {  
        "comment" : "Excellent literature",  
        "username" : "Tom",  
        "_id" : ObjectId("4e86f6ffffed0eb0be000001a")  
    }  
]
```

Let's try to retrieve reviews from "Gautam".

```
> db.books.find( { "reviews.username" : "Gautam" } )
```

What just happened?

The MongoDB classic act!

"reviews.username" searches inside all the elements in the array for any field called "username", which has the specified value.

Of course, there are other conventional ways of searching inside arrays.

Searching inside arrays using \$in and \$nin

This is something similar to the `IN` clause in SQL. Suppose we want to find documents for a specified number of values of a field, we can use the `$in` operator. Let's see one of our book objects:

```
> db.books.findOne()

{
  "_id" : ObjectId("4e86e45efed0eb0be0000010"),
  "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),
  "category_ids" : [
    ObjectId("4e86e4cbfed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
  ],
  "name" : "Oliver Twist",
}

}
```

We do know that these are `Category` objects referenced in some other collection. But that should not stop us from firing a direct query:

```
> db.books.find( { category_ids : { $in : [
    ObjectId("4e86e4cbfed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
  ] } } )
```

Alternatively, we could fire a `NOT IN` query too, as follows:

```
> db.books.find( { category_ids : { $nin : [
    ObjectId("55555555555555555555555555"),
    ObjectId("666666666666666666666666")
  ] } } )
```

This would return all the books in the collection!

Searching for exact matches using \$all

As we just saw `$in` helps us search for documents that have any one of the values in the array. It's `$all` that searches for documents that have all the values within the array in the field. Let's take this book object again:

```
> db.books.findOne()

{
  "_id" : ObjectId("4e86e45efed0eb0be0000010"),
  "author_id" : ObjectId("4e86e4b6fed0eb0be0000011"),
  "category_ids" : [
    ObjectId("4e86e4cbfed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
  ],
  "name" : "Oliver Twist",
}

}
```

Now, if we want to find books which belong to both the categories mentioned in the previous code, we fire the following query:

```
> db.books.find( { category_ids : { $all : [
    ObjectId("4e86e4cbfed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
] } } )
```

This will return all the books that are in both categories. However, unlike the earlier case of `$in`, the following query will not return the previously mentioned book because it doesn't belong to all the categories mentioned next:

```
> db.books.find( { category_ids : { $all : [
    ObjectId("4e86e4d9fed0eb0be0000011"),
    ObjectId("4e86e4d9fed0eb0be0000012"),
    ObjectId("4e86e4d9fed0eb0be0000013")
] } } )
```

Searching inside hashes

Just like arrays, we also want to search inside hashes. Searching inside hashes involves keys and values. Let's assume that the book object looks as follows (that is, a hash instead of an array):

```
{
  categories: {
    'drama': 1,
    'thriller': 2
  },
}
```

We can search for all books that have the `drama` set as 1:

```
> db.books.find({ "categories.drama" : 1 })
```

Notice that we access hash fields just like standard JSON object access.



It's interesting to note that the criteria for searching in hashes and arrays is the same in most cases.



Searching inside embedded documents

Searching inside embedded documents is exactly like searching inside hashes. This seems to make sense because MongoDB saves every document as a hash.



Embedded documents are sometimes also called nested documents in discussion.



The following is an example of an embedded document:

```
{
  "_id" : ObjectId("6234a68bfed0eb0beabcd234"),
  "name" : "The Adventures of Sindbad",
  "category" : {
    "_id" : ObjectId("5ad6f68bfed0eb0be1231213"),
    "name" : "Adventure",
  }
}
```

To fetch the `category` object it's exactly the same way as searching inside a hash:

```
> db.books.find( { "category.name" : "Adventure" } )
```

And just like that, searching inside arrays, hashes, and embedded documents have almost the same syntax!

Searching with regular expressions

The story isn't complete without regular expressions! Let's see a sample structure for the `names` collection:

```
{
  _id : ObjectId("1ad6f68bfed0eb0be1231234"),
  name : "Joe"
}
```

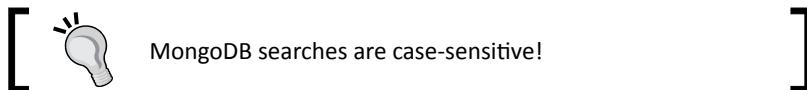
```
{  
  _id : ObjectId("1ad6f68bfed0eb0be1231235") ,  
  name : "Joey"  
}  
{  
  _id : ObjectId("1ad6f68bfed0eb0be1231236") ,  
  name : "Jonas South"  
}  
{  
  _id : ObjectId("1ad6f68bfed0eb0be1231237") ,  
  name : "Aron Bjoe"  
}
```

Time for action – using regular expression searches

Now if we want to search for all the objects that have Joe in their name, we can fire the following query:

```
> db.names.find({ name : /Joe/ } )  
  
{ _id : ObjectId("1ad6f68bfed0eb0be1231234") , name : "Joe" }  
{ _id : ObjectId("1ad6f68bfed0eb0be1231235") , name : "Joey" }
```

Notice that we got the objects that had a "Joe" in them. But wait! What happened to the third record, it has a Joe in it too!



Now, if we require all the names that have a joe in them, irrespective of the case, we fire a similar query again:

```
> db.names.find({ name : /joe/i } )  
  
{ _id : ObjectId("1ad6f68bfed0eb0be1231234") , name : "Joe" }  
{ _id : ObjectId("1ad6f68bfed0eb0be1231235") , name : "Joey" }  
{ _id : ObjectId("1ad6f68bfed0eb0be1231237") , name : "Aron Bjoe" }
```

Now we get all three objects. What if I want only the authors who start with a Jo, we fire another query as follows:

```
> db.names.find({ name : /^Jo/ } )  
  
{ _id : ObjectId("1ad6f68bfed0eb0be1231234") , name : "Joe" }  
{ _id : ObjectId("1ad6f68bfed0eb0be1231235") , name : "Joey" }  
{ _id : ObjectId("1ad6f68bfed0eb0be1231236") , name : "Jonas South" }
```

Notice the difference in the search result!

What just happened?

The magic of regular expressions! Here is a brief idea about how regular expressions work. Then we can try out something complicated.

Regular expressions are divided into two parts—pattern and occurrence. **Pattern**, as the name suggests, is the regular expression pattern. **Occurrence** is the number of times the pattern should occur:

Pattern	Occurrence
\w: Alphanumeric	a*: 0 or more of a
\d: Digits	a+: 1 or more of a
.: Any character	a?: 0 or 1 of a
\s: Any whitespace	a{10}: Exactly 10 of a
\W: Non alphanumeric	a{3,10}: between 3 and 10 of a
\D: Non digits	A{5,}: 5 or more of a
\S: Non whitespace	a{,10}: at most 10 of a
\b: Word boundary	[abc]: a or b or c
[a - z]: any character between a and z	[^abc]: not a, b or c
[0 - 9]: Any digit between 0 and 9	^: start of line
: regex separator	\$: end of line
	(...): regex group

While specifying the regular expressions, we write it entirely in front slashes (/):

```
/<some regex>/<flags>/
```

Flags can be:

- ◆ i: Case insensitive.
- ◆ m: Multiline.
- ◆ x: Extended—ignore all whitespaces in the regex.
- ◆ a: Dot all. Allow dot to match all characters, including new line characters!

Let's see examples of their usage:

For one or more occurrences of a:

```
/a+/
```

For one or more occurrences of a followed by 0 or more of b:

```
/a+b*/  
# abc or xyz only  
/abc|xyz/
```

For a case insensitive match for alphanumerics:

```
/\w/i
```

For zero or more occurrences of x,y or z:

```
/[xyz]*/
```

Have a go hero – validate an e-mail address

Build a regular expression to match an e-mail ID. Let's keep this simple and not strictly follow the ISO-compliant e-mail address format. This is just for learning and fun. Here are some hints:

- ◆ An e-mail ID should start with two alphabets
- ◆ An e-mail ID should be alphanumeric and may contain the following special characters such as ., +, and _

Some examples of valid e-mail IDs are `gautam@joshsoftware.com` and `gautam.rege@gmail.co.in` while those of invalid e-mail IDs are `gautam%rege@invalid` and `gautam.@[.com]`

Pop quiz – searching the right way

1. How do we find the 10th to 15th documents in the `books` collection, including the 10th and 15th document?
 - a. `db.books.find({}, {}, 10, 15)`
 - b. `db.books.find({}, {}, 10, 5)`
 - c. `db.books.find({}, {}, 6, 9)`
 - d. `db.books.find(10, 5)`
2. How do we find the books only with the `_id` and no other fields?
 - a. `db.books.find({}, { _id: 1})`
 - b. `db.books.find()`
 - c. `db.books.find({_id : 1})`
 - d. `db.books.find`

3. How can we find all the book documents that have a categories hash in them?
 - a. db.books.find(\$exists: { categories : 1 })
 - b. db.books.find({ categories: \$exists })
 - c. db.books.exists({ categories: 1 })
 - d. db.books.find({ categories : { \$exists : 1 } })
4. How do we find all the books whose title do not have the words the or a in it? For example, "The Great Escape" should not be selected but "Tale of Two Cities" should be selected.
 - a. db.books.find({ \$nin: { title : [/the/, /a/] } })
 - b. db.books.find({ title: { \$nin : [/the\b/i, /a\b/i] } })
 - c. db.books.find({ title: { \$ne : "the"}, { \$ne : "a"} })
 - d. db.books.find({ title: { \$neq : /the|a/i } })

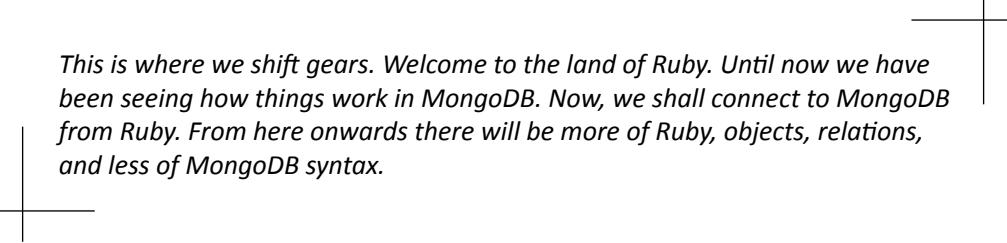
Summary

In this chapter, we have seen the various ways to query objects in MongoDB. We can search by fields, inside arrays, hashes, and even embedded objects. We can even search by regular expressions. Searching forms a vital part of any application as there would typically be a lot more reads than writes to the database. Searching efficiently improves the performance of the application, so it's important that we understand these concepts well.

This is just the tip of the iceberg. In the next chapters, we shall relate these querying paradigms via Ruby using the various Ruby DataMappers.

5

Ruby DataMappers: Ruby and MongoDB Go Hand in Hand



This is where we shift gears. Welcome to the land of Ruby. Until now we have been seeing how things work in MongoDB. Now, we shall connect to MongoDB from Ruby. From here onwards there will be more of Ruby, objects, relations, and less of MongoDB syntax.

In this chapter we shall learn the following:

- ◆ Why we need Ruby DataMappers
- ◆ The different Ruby DataMappers and the power of open source
- ◆ Comparing different Ruby DataMappers
- ◆ Querying objects
- ◆ Managing object relations

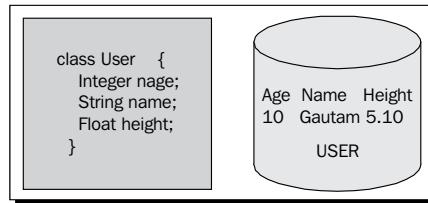
Let's dive straight into Ruby with our Sodibee library management system!

Why do we need Ruby DataMappers

Well, how else would we connect to MongoDB? Let's first see what a data mapper is.

By definition, a **datamapper** is a process, framework, or library that maps two different sources of data. In our particular case, one source is the MongoDB data structure and the other is the Ruby object model.

If we have a relational database, we have tables which have columns. These are often mapped to the object-oriented language constructs—classes map to tables and attributes map to columns. Considering the object-oriented nature of Ruby and the document data structure of MongoDB, this makes a very good combination for a DataMapper. A class maps to the collection name and the object is the document inside a collection. This is shown in the following diagram:



Instead of directly firing queries on MongoDB using raw connections, it's better to have an abstraction—via a data mapper. As is common in the open source world, there are usually multiple options available for everything and Ruby DataMappers are no different. There are plenty of Ruby DataMappers for MongoDB and more are being born. In this book, we shall concentrate on a few of the most popular ones.

The mongo-ruby-driver

This is the core driver that is available via the `mongo` gem. To install this gem, we simply use the following command:

```
$ gem install mongo
```

MongoDB uses Binary JSON (BSON) to save data. So it's also necessary to install `bson` and `bson_ext` gems. In most cases, as these are dependent gems, they should install along with the `mongo` gem. Remember that you require the same version for `mongo`, `bson`, and `bson_ext`! At the time of writing this book, the latest version of this driver is 1.6.2.

In case you see messages like the one shown next, please ensure that `bson`, `bson_ext`, and `mongo` gem have the same version:

```
**Notice: C extension not loaded. This is required for optimum MongoDB  
Ruby driver performance.
```

```
You can install the extension as follows:  
gem install bson_ext
```

```
If you continue to receive this message after installing, make sure  
that the bson_ext gem is in your load path and that the bson_ext and  
mongo gems are of the same version.
```

```
$
```

Time for action – using mongo gem

It's never complete without an example. So, let's write a sample Ruby program to connect to our Sodibee database.

```
require 'mongo'

conn = Mongo::Connection.new
db = conn['sodibee_development']
coll = db['books']

puts coll.find.first.inspect
```

The output should look something like this:

```
$ ruby mongo_driver.rb
```

```
{"_id"=>BSON::ObjectId('4e86e45efed0eb0be0000010'), "author_id"=>BSON::Object
Id('4e86e4b6fed0eb0be0000011'), "category_ids"=>[BSON::ObjectId('4
e86e4cbfed0eb0be0000012'), BSON::ObjectId('4e86e4d9fed0eb0be0000013')], "name"=>"Oliver Twist", "published_on"=>2002-12-30 00:00:00 UTC, "publisher"=>"Dover Publications", "reviews"=>[{"_id"=>BSON::ObjectId(
'4e86f68bfed0eb0be0000018'), "comment"=>"wow!", "username"=>"Gautam"}, {"comment"=>"Excellent literature", "username"=>"Tom", "_id"=>BSON::Object
Id('4e86f6ffffed0eb0be000001a')}], "votes"=>[{"username"=>"Gautam", "rating"=>3}]}
```

What just happened?

Wow! We just connected to MongoDB from a Ruby program and fetched the first book from the books collection. Let's take this slowly, shall we? Let's see the previous code again:

```
require 'mongo'

conn = Mongo::Connection.new
db = conn['sodibee_development']
coll = db['books']

puts coll.find.first.inspect
```

The command require loads the Ruby Mongo library.

 In case you are using Ruby 1.8.7, you may need to require "rubygems" or add "rubygems" to your RUBYOPTS environment variable. In Ruby 1.9 onwards, this is implicitly included. Rubygems is a gem which helps Ruby load Ruby library paths.

Let's have a look at the previous code once again:

```
require 'mongo'

conn = Mongo::Connection.new
db = conn['sodibee_development']
coll = db['books']

puts coll.find.first.inspect
```

This sets up the connection with MongoDB. Did I hear you say "What the hell?! Magically? what happened to the host or the port?" Welcome to the world of "convention over configuration".

The Mongo driver is configured with defaults:

- ◆ Host: Localhost is the default
- ◆ Port: 27017 is the default
- ◆ Options:
 - safe: If it is true, MongoDB starts in safe mode (it is false by default)
 - slave_ok: It is (false by default) set to true only when connecting to a single slave
 - logger: Remember that logging can degrade performance (It is nil by default)
 - pool_size: It is (1 by default) the number of sockets connections in the pool
 - pool_timeout: It is (5.0 seconds by default) the seconds to wait before which an exception will be thrown
 - op_timeout: It is (nil by default) the read timeout. There is no timeout by default
 - connect_timeout: It is (nil by default) the connection timeout. By default the connection never times out
 - ssl: It is (false by default) set to true for secure connections only

Whoa! These are a lot of options. Notice the default values. You don't need to remember them all if you are working with defaults.

Once again, let's have a look at the previous code:

```
require 'mongo'

conn = Mongo::Connection.new
```

```

db = conn['sodibee_development']
coll = db['books']

puts coll.find.first.inspect

```

We now select the database we require and the collection we want.

Guess what, looks are deceptive! The `Mongo::Connection` class has the method `Mongo::Connection#[]` that initializes a `Mongo::Db` object and returns it. We can then access the collection we want in this database. In case you require some specific options for the database object (for example, you may want to access the database in `strict` mode), you would need to explicitly instantiate the database object. This is done as follows:

```
db = Mongo::Db.new('sodibee_development', conn, :strict => true)
```



Strict mode ensures that the collection exists before accessing it.
Otherwise it throws an error.



Of course, we usually require the former:

```

require 'mongo'

conn = Mongo::Connection.new
db = conn['sodibee_development']
coll = db['books']

puts coll.find.first.inspect

```

The command `coll.find` gets us the collection object cursor (similar to database cursors) and from this we print the first. We shall see a lot of the `find` method later on in this chapter.

The Ruby DataMappers for MongoDB

We do not want to get into details of how the mongo-ruby-driver is written. This is because it does a lot of work under the cover and we don't want to get our hands that dirty! Think of this like a device driver—we use them but we are not the experts who write them. So, we leave the nitty-gritty details to the DataMappers!

There are quite a few DataMappers built in Ruby to map to documents in MongoDB. The ones that are very popular while this book is being written, are:

- ◆ `MongoMapper`
- ◆ `Mongoid`

We shall now learn how to use both and you can see for yourself which to use. It's a close race for the winner and towards the end of this chapter I do declare a verdict based on my experiments with them.

MongoMapper

MongoMapper was one of the first Ruby data mappers for MongoDB. Created by John Nunemaker in early 2009, it has gained a lot of popularity. The entire library is written in Ruby. However, the MongoMapper is tightly coupled for Rails applications and does not use the mongo-ruby-driver.

Mongoid

The work for the mongo-ruby-driver began in late 2008 and as it got stable it was also heavily used in Ruby DataMappers. Mongoid, which began in mid-2009 by Durran Jordan has gained tremendous popularity. It uses the Mongo driver for accessing MongoDB.

There has not been any clear winner among them, but my preference is with Mongoid. I do leave it to your choice which one to choose as I will be going through both of them in some detail.

Setting up DataMappers

We have seen how we can use the mongo-ruby-driver to access the MongoDB store via Ruby. Now, we shall see how to use DataMappers for connecting, creating, and querying documents.

Configuring MongoMapper

As with any gem installation, this is done as follows:

```
$ gem install mongo_mapper
```

If you are using Bundler, we could also set this in the Gemfile using the following:

```
gem 'mongo_mapper'
```

If you are using Rails 3.1 or greater, we can create a new Rails project as follows:

```
$ rails new sodibee-mm
```

You should see something as follows:

```
create
  create  README
  create  Rakefile
```

```
create config.ru
create .gitignore
create Gemfile

create vendor/plugins
create vendor/plugins/.gitkeep
run bundle install

Fetching source index for http://rubygems.org/
Using rake (0.9.2.2)
Using multi_json (1.0.4)
...
Installing sqlite3 (1.3.5) with native extensions
Installing turn (0.8.2)
Installing uglifier (1.2.0)
Your bundle is complete! Use 'bundle show [gemname]' to see where a
bundled gem is installed.

$
```

Now that we have set up a project, we need to install MongoMapper.

Time for action – configuring MongoMapper

Let's set up MongoMapper for generating the mongo config file.

```
$ rails generate mongo_mapper:config
      create config/mongo.yml
```

The contents of config/mongo.yml look like the following code listing:

```
defaults: &defaults
  host: 127.0.0.1
  port: 27017

development:
  <<: *defaults
  database: sodibee_mm_development

test:
  <<: *defaults
  database: sodibee_mm_test
```

```
# set these environment variables on your prod server
production:
  <<: *defaults
  database: sodibee_mm
  username: <%= ENV['MONGO_USERNAME'] %>
  password: <%= ENV['MONGO_PASSWORD'] %>
```

The preceding file is a standard YML file with defaults. Now let's generate a mongo model as follows:

```
$ rails generate mongo_mapper:model Author
```

The preceding code should generate the following files:

```
create app/models/author.rb
invoke test_unit
create test/unit/author_test.rb
create test/fixtures/authors.yml
```

The model file would be like the following—very complicated!

```
class Author
  include MongoMapper::Document

end
```

What just happened?

We just saw two things:

- ◆ We configured MongoMapper (through config/mongo.yml).
- ◆ We generated models pre-configured with MongoMapper

MongoMapper::Document is a Ruby module that we can include in any model. Rails 3 now advocates the use of ActiveRecord and not inheritance from ActiveRecord.



Ruby module mixins are a unique and interesting feature of Ruby. Using modules, we can make classes richer by including or extending modules in classes.



Have a go hero – creating models using MongoMapper

Create the other Sodibee models for MongoMapper: book, category, and review. Refer to *Chapter 2, Diving Deep into MongoDB* for details on these fields.

Configuring Mongoid

Just like MongoMapper, Mongoid can be installed as a gem as follows:

```
$ gem install mongoid
```

You can also put the following in a Gemfile:

```
gem 'mongoid'
```

Time for action – setting up Mongoid

Once we have a project created (just like we saw earlier), we can configure Mongoid as follows:

```
$ rails generate mongoid:config  
create config/mongoid.yml
```

The next code listing is what the config/mongoid.yml looks like:

```
development:  
  host: localhost  
  database: sodibee_development  
  
test:  
  host: localhost  
  database: sodibee_test  
  
# set these environment variables on your prod server  
production:  
  host: <%= ENV['MONGOID_HOST'] %>  
  port: <%= ENV['MONGOID_PORT'] %>  
  username: <%= ENV['MONGOID_USERNAME'] %>  
  password: <%= ENV['MONGOID_PASSWORD'] %>  
  database: <%= ENV['MONGOID_DATABASE'] %>  
  # slaves:  
  #   - host: slave1.local  
  #     port: 27018  
  #   - host: slave2.local  
  #     port: 27019
```

There is no direct generator for Mongoid. Simply do the following:

```
class Author  
  include Mongoid::Document  
  
end
```

Your Rails project should not load ActiveRecord (For Rails version less than 3.0).

Ensure the following:

- ◆ Remove config/database.yml
- ◆ Remove the following line from config/application.rb:

```
require 'rails/all'
```
- ◆ Add the following line in config/application.rb:

```
require "action_controller/railtie"
require "action_mailer/railtie"
require "active_resource/railtie"
require "rails/test_unit/railtie"
```

For Rails 3.1.x and Rails 3.0.x to ensure that you do not load ActiveRecord.

Execute the following command:

```
$ rails new <project_name> -O -skip-bundle
```

What just happened?

We set up Mongoid, which looks almost similar to MongoMapper. However, the Mongoid::Document and MongoMapper::Document differ considerably in the way they are structured internally.

MongoMapper::Document includes the various plugins as follows:

- ◆ include Plugins::ActiveModel
- ◆ include Plugins::Document
- ◆ include Plugins::Querying
- ◆ include Plugins::Associations
- ◆ include Plugins::Caching
- ◆ include Plugins::Clone
- ◆ include Plugins::DynamicQuerying
- ◆ include Plugins::Equality
- ◆ include Plugins::Inspect
- ◆ include Plugins::Indexes
- ◆ include Plugins::Keys
- ◆ include Plugins::Dirty
- ◆ include Plugins::Logger

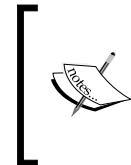
- ◆ include Plugins::Modifiers
- ◆ include Plugins::Pagination
- ◆ include Plugins::Persistence
- ◆ include Plugins::Accessible
- ◆ include Plugins::Protected
- ◆ include Plugins::Rails
- ◆ include Plugins::Safe
- ◆ include Plugins::Sci
- ◆ include Plugins::Scopes
- ◆ include Plugins::Serialization
- ◆ include Plugins::Timestamps
- ◆ include Plugins::Userstamps
- ◆ include Plugins::Validations
- ◆ include Plugins::EmbeddedCallbacks
- ◆ include Plugins::Callbacks

Mongoid::Document includes these modules via Mongoid::Components as follows:

- ◆ include ActiveRecord::Conversion
- ◆ include ActiveRecord::MassAssignmentSecurity
- ◆ include ActiveRecord::Naming
- ◆ include ActiveRecord::Observing
- ◆ include ActiveRecord::Serializers::JSON
- ◆ include ActiveRecord::Serializers::Xml
- ◆ include Mongoid::Atomic
- ◆ include Mongoid::Attributes
- ◆ include Mongoid::Collections
- ◆ include Mongoid::Copyable
- ◆ include Mongoid::DefaultScope
- ◆ include Mongoid::Dirty
- ◆ include Mongoid::Extras
- ◆ include Mongoid::Fields
- ◆ include Mongoid::Hierarchy

- ◆ include Mongoid::Indexes
- ◆ include Mongoid::Inspection
- ◆ include Mongoid::JSON
- ◆ include Mongoid::Keys
- ◆ include Mongoid::Matchers
- ◆ include Mongoid::NamedScope
- ◆ include Mongoid::NestedAttributes
- ◆ include Mongoid::Persistence
- ◆ include Mongoid::Relations
- ◆ include Mongoid::Safety
- ◆ include Mongoid::Serialization
- ◆ include Mongoid::Sharding
- ◆ include Mongoid::State
- ◆ include Mongoid::Validations
- ◆ include Mongoid::Callbacks
- ◆ include Mongoid::MultiDatabase

If we compare the modules, there is little to debate. Both have similar features but are implemented in different ways internally. The only way to understand them in detail is to dig into the code.



Initially, I did wonder about why MongoMapper and Mongoid don't just merge like Rails and Merb. When I started digging into the code, I realized how different the internal implementation is. Do read this <http://www.rubyinside.com/mongoid-vs-mongomapper-two-great-mongodb-libraries-for-ruby-3432.html>.

Creating, updating, and destroying documents

Now let's work with objects—creating, updating, and deleting them. But first, we need to set up the model with attributes. We add these attributes in the models directly. Each attribute has a name and also specifies the type of data storage. To ensure we see all the standard data types, we shall see the `Person` model.

Defining fields using MongoMapper

We define the model in the `app/models/person.rb` file as follows:

```
class Person
  include MongoMapper::Document

  key :name,      String
  key :age,       Integer
  key :height,    Float
  key :born_on,   Date
  key :born_at,   Time
  key :interests, Array
  key :is_alive,  Boolean
end
```

Defining fields using Mongoid

With Mongoid, there is just a difference in syntax:

```
class Person
  include Mongoid::Document

  field :name,      type: String
  field :age,       type: Integer
  field :height,    type: Float
  field :born_on,   type: Date
  field :born_at,   type: Time
  field :interests, type: Array
  field :is_alive,  type: Boolean
end
```

Creating objects

The way to create objects does not depend on the mapper. Just like we create objects in Ruby, we pass the parameters as hash arguments.

Time for action – creating and updating objects

Let's create an object of the Person model with different values as shown next:

```
person = Person.new( name: "Tom Sawyer", age: 33, height: 5.10,
                     born_on: Date.parse("1972-12-23"),
                     born_at: Time.now, is_alive: true,
                     interests: ["Soccer", "Movies"])

=> #<Person _id: BSON::ObjectId('4ef4ab59fed0eb8962000002'), age: 33,
     born_at: Fri, 23 Dec 2011 16:24:57 UTC +00:00, born_on: Sat, 23 Dec
     1972, height: 5.1, interests: ["Soccer", "Movies"], is_alive: true,
     name: "Tom Sawyer">
```

Now, if we want to update the previous object, we save it by calling the `save` method after setting the name. It is done as follows:

```
person.name = "Huckleberry Finn"  
person.save
```

Now if we want to destroy this object, we simply issue the following command:

```
person.destroy
```

That's it!

What just happened?

There is no different syntax when using Mongoid or MongoMapper. This is the real advantage of using Ruby DataMappers.

In reality, Ruby frameworks such as Rails and Sinatra, try to be as independent of the data source as possible. So, if we used MySQL, PostgreSQL, or any other database, we can easily migrate them to MongoDB and vice versa by altering some part of the code.

However, this does not mean that there would be no code change. As we will soon see in the querying documents, and later in *Understanding model relationships*, it's not that simple and straightforward.

Using finder methods

This is where the real fun begins! We shall start seeing different ways to search among objects. Both, MongoMapper and Mongoid try to adhere to the standard querying interface as much as possible.

Finders are routines that return the objects as part of the result. Both MongoMapper and Mongoid implement the standard querying interface.

Using find method

The `find` method finds the object with the specified ID:

```
person = Person.find('4ef4ab59fed0eb8962000002')
```



It's interesting to see that the MongoDB object ID is `_id` while for Ruby it is `id`. Both can be used interchangeably.



Using the first and last methods

As the name suggests, we can get the first and the last objects with these methods as follows:

```
Person.first    # => The first object.  
Person.last    # => The last object.
```

Using the all method

As the name suggests, this method fetches all the objects. We can optionally pass it some selection criteria too. This is done as follows:

```
Person.all
```

Or

```
Person.all(:age => 33)
```

So, what happens if we have 1 million person objects and we fire `Person.all`? Does this mean all 1 million objects are fetched? MongoDB internally uses the cursor to fetch objects in batches. By default 1000 objects are fetched.

Using MongoDB criteria

Criteria are proxy objects or intermediate results. These are not queries that are fired on the database immediately—that is why they are called the criteria. We can chain criteria. When all criteria are completed and we really need the data, the final query is fired and documents are fetched from the database. This has immense advantages while programming in Ruby.



In Rails, these are called scopes (and in earlier versions they were called named scopes).



We saw the use of `all` earlier. Mongoid treats `all` as a criteria while MongoMapper resolves it—that is `all` returns an array.

Executing conditional queries using where

This is the most frequently used criterion:

```
Person.where(:all => 33)
```

This looks uncannily similar to the `all` method we have seen earlier. However, the result from `where` is entirely different from `all`.

Time for action – fetching using the where criterion

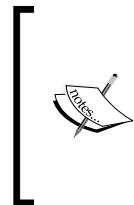
When we want to fetch (and chain) results, we use the `where` criteria. For example, if we have a web application and there are different filters, such as age and name, we can chain these criteria easily in a Ruby application as shown next:

```
people = Person.where(:age.gt => 15)
people = people.where(:name => /saw/i)

=> #<Person _id: BSON::ObjectId('4ef4ab59fed0eb8962000002'), age: 33,
born_at: Fri, 23 Dec 2011 16:24:57 UTC +00:00, born_on: Sat, 23 Dec
1972, height: 5.1, interests: ["Soccer", "Movies"], is_alive: true,
name: "Tom Sawyer">
```

What just happened?

We not only saw how criteria work but also the different selection criteria syntax. Let's analyze this in detail.



MongoMapper uses Plucky—a gem for managing proxy objects. It basically creates a lambda based on the selection criteria. Then we can chain these lambda instances together and get a result. This same functionality in Mongoid is available in the `Mongoid::Criteria` object. This is one of the key internal differences between both MongoMapper and Mongoid.

Take a look at the following code:

```
people = Person.where(:age.gt => 15)
people = people.where(:name => /saw/i)
```

The previous code returns a criterion object. If we are using MongoMapper, this would return a Plucky object:

```
=> #<Plucky::Query age: {"$gt"=>15}, transformer: #<Proc:0x1d8cab0@/Users/gautam/.rvm/gems/ruby-1.9.2-p290/gems/mongo_mapper-0.10.1/lib/mongo_mapper/plugins/querying.rb:79 (lambda) >>
```

If we use Mongoid, the following code would return a `Mongoid::Criteria` object:

```
=> #<Mongoid::Criteria
selector: {},
options: { :age=>{"$gt"=>15}},
class: Person,
embedded: false>
```

It's important to remember that the database query has not been fired yet.



Notice the construct `:age.gt => 15`. This is the short form of writing `:age => { '$gt' => 15 }` and this means "age greater than 15".



Now let's analyze the next line. This makes things very interesting!

```
people = Person.where(:age.gt => 15)
people = people.where(:name => /saw/i)
```

The `people` criterion is now "chained" with another criterion. If we use MongoMapper, this is what we see of the `people` object now:

```
=> #<Plucky::Query age: {"$gt"=>15}, name: /saw/i, transformer:
#<Proc:0x1d86778@/Users/gautam/.rvm/gems/ruby-1.9.2-p290/gems/mongo_
mapper-0.10.1/lib/mongo_mapper/plugins/querying.rb:79 (lambda) >>
```

Did you notice the second line of code:

```
people = people.where(:name => /saw/i)
```

We have chained `where` to the earlier `people` criterion. Also notice that `name: /saw/i` is now part of the selection criterion. If we use Mongoid, this would look like the following:

```
=> #<Mongoid::Criteria
  selector: { :age=>{"$gt"=>15}, :name=>/saw/i},
  options: {},
  class: Person,
  embedded: false>
```

It's interesting to know that the query has still not been fired. Only when all the criteria are fulfilled, will the objects be fetched from the database. This is unlike an SQL query, which directly fetches results; this is instead more efficient as we resolve the entire scope of the selection before fetching objects.



Notice the `/saw/i` construct. This is a case-insensitive regular expression search for any name that has `saw` in it, such as Sawyer!



Revisiting limit, skip, and offset

We have seen the use of `limit`, `skip`, and `offset` earlier in *Chapter 4, Working Out Your Way with Queries*. Now, we shall see how simple it is to set them from MongoMapper or Mongoid. It is done as follows:

```
Person.where(:age.gt => 15).limit(5)
```

Pagination is an excellent example of this. This chains criteria to ensure that at most five results are returned in the results set.

```
Person.all.skip(5).limit(5) # Page 2 with 5 elements  
Person.all.skip(10).limit(5) # Page 3 with 5 elements
```

Understanding model relationships

Now we shall see different types of object relations. They are as follows :

- ◆ One-to-many relation
- ◆ Many-to-many relation
- ◆ One-to-one relation
- ◆ Polymorphic relations

The one to many relation

Let's get back to Sodibee! Let's assume that one book has one author. In a relationship statement, this means, "An Author has many books" and "A book belongs to one author". We write a relationship exactly like this.

Time for action – relating models

We shall see how we can set up relations in both MongoMapper as well as Mongoid.

Using MongoMapper

As we know the author model is in the `app/models/author.rb` file and book is in the `app/models/book.rb` file:

```
class Author  
  include MongoMapper::Document  
  
  key :name,      String  
  
  many :books  
  
end  
  
class Book  
  include MongoMapper::Document  
  
  key :name,      String
```

```
key :publisher,      String
key :published_on,  Date

belongs_to :author

end
```

Using Mongoid

The file locations remain the same, it's only the syntax that changes as follows:

```
class Author
  include Mongoid::Document

  field :name, type: String

  has_many :books

end

class Book
  include Mongoid::Document

  field :name, type: String
  field :publisher, type: String
  field :published_on, type: Date

  belongs_to :author

end
```

Let's now create some books and authors. This object creation code remains the same, irrespective of which data mapper we use. We create books and authors as follows:

```
irb> charles = Author.create(name: "Charles Dickens")
=> => #<Author _id: BSON::ObjectId('4ef5a7eafed0eb8c7d000001'), name: "Charles Dickens">

irb> b = Book.create (name: "Oliver Twist", published_on: Date.parse("1983-12-23"), publisher: "Dover Publications", author: charles)
=> #<Book _id: BSON::ObjectId('4ef5a888fed0eb8c7d000002'), author_id: BSON::ObjectId('4ef5a7eafed0eb8c7d000001'), name: "Oliver Twist", published_on: Fri, 23 Dec 1983, publisher: "Dover Publications">
```

What just happened?

`many` is a method in MongoMapper that takes the relation (also called the association) as a parameter. Its equivalent in Mongoid is `has_many`.

`belongs_to` is a reverse relation that tells us who the parent is.

As with all relations, the child references the parent. This means the book document has an `author_id` field.



In SQL, it's a thumb rule that the foreign key resides with the child table.
Similarly, the reference resides in the child document in MongoDB.



Let's look at the book creation code in more detail:

```
irb> b = Book.create (name: "Oliver Twist", published_on: Date.  
parse("1983-12-23"), publisher: "Dover Publications", author: charles)  
  
=> #<Book _id: BSON::ObjectId('4ef5a888fed0eb8c7d000002'), author_id:  
BSON::ObjectId('4ef5a7eafed0eb8c7d000001'), name: "Oliver Twist",  
published_on: Fri, 23 Dec 1983, publisher: "Dover Publications">
```

Notice, that we have passed `author: charles`, a variable which references the author object. However, when the object is created we see `author_id: BSON::ObjectId(..)`

The many-to-many relation

Let's introduce the `Category` model here. A book can have many categories and a category can have many books.

Time for action – categorizing books

As always, we shall now see how MongoMapper achieves a many-to-many relation first and then how Mongoid does the same.

MongoMapper

We are adding a new model—`app/models/category.rb`. This is done as follows:

```
class Category  
  include MongoMapper::Document  
  
  key :name, String  
  key :book_ids, Array
```

```

many :books, in: :book_ids

end

class Book
  include MongoMapper::Document

  key :title,      String
  key :publisher,   String
  key :published_on, Date

  belongs_to :author

end

```

Mongoid

The following code shows how we do this using Mongoid:

```

class Category
  include Mongoid::Document

  key :name,  String

  has_and_belongs_to_many :books

end

class Book
  include MongoMapper::Document

  key :title,      String
  key :publisher,   String
  key :published_on, Date

  belongs_to :author
  has_and_belongs_to_many :categories

end

```

Here is another area where MongoMapper and Mongoid differ in the internal implementation. Notice, that when using MongoMapper, the Book model has no changes. This means we cannot access the categories of a book from the Book object directly. We shall see this in more detail.



MongoMapper has only a one-way association for many-to-many. Mongoid maintains the inverse relation, that is, it updates both documents. A plus one for Mongoid!

Accessing many-to-many with MongoMapper

First create a few categories as follows:

```
irb> fiction = Category.create(name: "Fiction")
=> #<Category _id: BSON::ObjectId('4ef5b159fed0eb8d9c00000a'), book_ids: [], name: "Fiction">

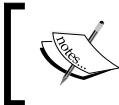
irb> drama = Category.create(name: "Drama")
=> #<Category _id: BSON::ObjectId('4ef5b231fed0eb8df5000005'), book_ids: [], name: "Drama">
```

Now, let's associate our book with these categories as follows:

```
irb> fiction.books << Book.first
irb> fiction.save!
```

So far so good! We should be able to retrieve this relation too. This is done as shown next:

```
irb> fiction.books
=> [#<Book _id: BSON::ObjectId('4ef5a888fed0eb8c7d000002'), author_id: BSON::ObjectId('4ef5a7eafed0eb8c7d000001'), name: "Oliver Twist", published_on: Fri, 23 Dec 1983, publisher: "Dover Publications">]
```



In MongoMapper, we cannot find the categories of a book object. We have to look via the Category model only, as the inverse relation is not supported yet.



Accessing many-to-many relations using Mongoid

Let's create a few categories again as follows:

```
irb> fiction = Category.create(name: "Fiction")
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name: "Fiction", book_ids: []>

irb> drama = Category.create(name: "Drama")
=> #<Category _id: 4e86e4d9fed0eb0be0000013, _type: nil, name: "Drama", book_ids: []>
```

Notice the `book_ids` attribute. It is present because of the `has_and_belongs_to_many` statement. Now let's associate the books and categories as follows:

```
irb> fiction.books << Book.first
```

That's it! Now let's check the relation by fetching it as follows:

```
irb> fiction.books.first
=> => #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", published_on: 2002-12-30 00:00:00
UTC, author_id: BSON::ObjectId('4e86e4b6fed0eb0be0000011'), category_ids: [BSON::ObjectId('4e86e4cbfed0eb0be0000012')], name: "Oliver
Twist">
```

Looks good! However, let's go one step further than MongoMapper.

```
irb> Book.first.categories
=> [#<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:
"Fiction", book_ids: [BSON::ObjectId('4e86e45efed0eb0be0000010')]> ]
```

What just happened?

I would give this round to Mongoid. We created many-to-many relations in both MongoMapper and Mongoid. However, Mongoid maintains the inverse relation!

So, if we were using MongoMapper, the following relation gives an error:

```
irb> Book.first.categories
NoMethodError: undefined method 'categories' for #<Book:0x1d63fd4>
from:      (method_missing)
```

This would not happen if we were using Mongoid.

 When we write many :books in the model, the many method defines a new method called books, which references the association. As the many-to-many relation is one-sided in MongoMapper, we have not declared any association in the book model for categories. Hence, the method_missing error.

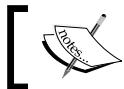
 One additional point to be mentioned here is that in MongoMapper, we save information to an array, not a relation. So, the object has to be explicitly saved. In Mongoid, we use an association to save the relation, so we do not need to call save explicitly on the object.

The one-to-one relation

Let's add a BookDetail model to Sodibee. The BookDetail model contains information about the number of pages, the cost, the binding style, among others.

Using MongoMapper

We will now add the new model app/models/book_detail.rb.



In Rails, the BookDetail model is stored in the book_detail.rb file—snake case.



We can add the BookDetail model using MongoMapper as follows:

```
class Book
  include MongoMapper::Document

  key :title,          String
  key :publisher,      String
  key :published_on,   Date

  belongs_to :author
  one :book_detail

end

class BookDetail
  include MongoMapper::Document

  key :page_count, Integer
  key :price,       Float
  key :binding,     String
  key :isbn,        String

  belongs_to :book

end
```

Using Mongoid

Now we will extend the book model and add the new book_detail.rb as follows:

```
class Book
  include Mongoid::Document

  key :title,          String
  key :publisher,      String
  key :published_on,   Date

  belongs_to :author
  has_and_belongs_to_many :categories
  has_one :book_detail
```

```

end

class BookDetail
  include Mongoid::Document

  field :page_count, type: Integer
  field :price, type: String
  field :binding, type: String
  field :isbn, type: String

  belongs_to :book

end

```

Time for action – adding book details

Let's add book details for our book now. It's the same for both MongoMapper and Mongoid. The following code shows you how to do it:

```

irb> oliver = Book.first
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", published_on: 2002-12-30 00:00:00
UTC, author_id: BSON::ObjectId('4e86e4b6fed0eb0be0000011'), category_ids: [BSON::ObjectId('4e86e4cbfed0eb0be0000012')], name: "Oliver
Twist">
irb> oliver.create_book_detail(page_count: 250, price: 10, binding:
"standard", isbn: "124sdf23sd")
=> => #<BookDetail _id: 4ef5bdaafed0eb8ed7000002, _type: nil, page_
count: 250, price: 10.0, binding: "standard", isbn: "124sdf23sd",
book_id: BSON::ObjectId('4e86e45efed0eb0be0000010')>

```

What just happened?

We created a `BookDetail` object. That was obvious, wasn't it? However, a closer look at this and we learn something new as follows:

```
irb> oliver.create_book_detail(page_count: 250, price: 10,
```

When we have only a direct single association (or relation), we build it using the `create_` prefix. In the earlier case for a many-to-many relation, in case we want to add a new category, we could do something similar to the following:

```
irb> oliver.categories.create(name: "New Theater")
```

This would create a new category and associate that category with the `Book` object.

Have a go hero – create the other models

Create the Book, Author, and Category objects. Then associate them!

Understanding polymorphic relations

Before we even see how this is done using MongoMapper or Mongoid, it's important to understand the basic concept of polymorphic relations.

Polymorphic means multiple forms or multiple behaviors. When we use it in the context of a database, we do mean multiple forms of the object. Let's see an example.

"Abstract base objects" in technical terms and "Generic common nouns" in layman's terms are ideal examples for explaining polymorphic relations.

For example, a vehicle could mean a two-wheeler, three-wheeler, a car, a truck or even a space shuttle! A vehicle has at least one driver, so we have a relation between a vehicle and its driver. Let's assume that a vehicle has only one driver. A driver has different skills. For example he could be a cyclist, an astronaut, or an F1 driver! So, how do we map these different types of driver profiles?

Implementing polymorphic relations the wrong way

If we are using a relational database, we can create a table called vehicles. We map all attributes of a vehicle as columns in the table. So, we have all fields of a vehicle (right from a cycle to a space shuttle) mapped in columns and then populate only the relevant fields. We also keep a type column, which signifies what the vehicle type is—cycle, car, space shuttle among others.

This is crazy because we could end up with a table having a few thousand columns! Wrong, wrong, wrong!

You could argue that using a document database like MongoDB could alleviate this problem — because it is schema free. So, we could create a collection called vehicles and we could map different fields in a document and keep going until we can. The type field identifies the type of the vehicle. However, this is still not a practical or a scalable approach and degrades performance as data increases. Considering that a document has a limited size.

Implementing polymorphic relations the correct way

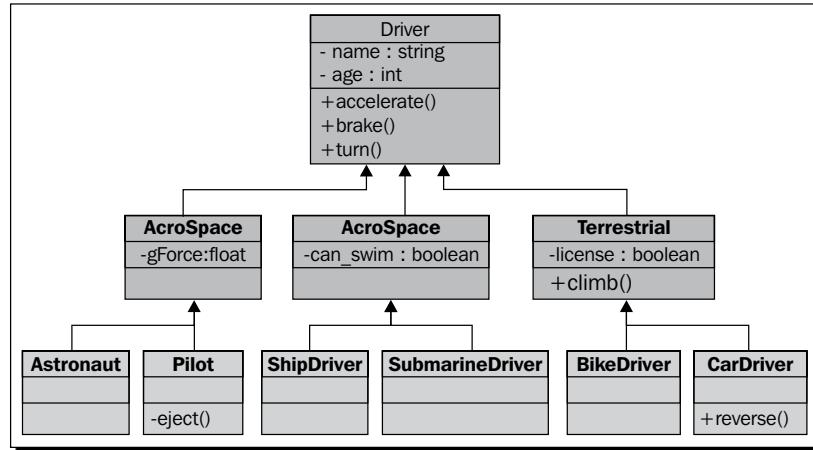
There are two types of polymorphic relations:

- ◆ **Single Collection Inheritance (SCI)**
- ◆ Basic polymorphic relations

We shall study both of them in detail. After that, we shall see when to choose the right approach. Let's study them first.

Single Collection Inheritance

This is very similar to the inheritance of standard object-oriented programming. See the following diagram for the inheritance hierarchy for drivers:



Time for action – managing the driver entities

Let's see the code for this. First let's create the generic Driver model as follows:

```

# app/model/driver.rb
class Driver
  include Mongoid::Document

  field :name, type: String
  field :age, type: Integer
  field :address, type: String
  field :weight, type: Float

end
  
```

This is pretty much straightforward. Now let's see the AeroSpace, Terrestrial, and Marine classes. They are shown next:

```

# app/models/terrestrial.rb
class Terrestrial < Driver
  field :license, type: Boolean
  
```

```
end

# app/models/marine.rb
class Marine < Driver
  field :can_swim, type: Boolean
end

# app/model/aero_space.rb
class AeroSpace < Driver
  field :gforce, type: Float
end
```

Here we simply inherit from the `Driver` class. Let's dive deeper. Let's create the `Pilot`, `Astronaut`, and other lower-level classes as follows:

```
# app/models/pilot.rb
class Pilot < AeroSpace
end

# app/models/astronaut.rb
class Astronaut < AeroSpace
end

# app/models/ship_driver.rb
class ShipDriver < Marine
end

# app/models/submarine_driver.rb
class SubmarineDriver < Marine
end

# app/models/car_driver.rb
class CarDriver < Terrestrial
end

# app/models/bike_driver.rb
class BikeDriver < Terrestrial
end
```

Now let's create some objects as follows:

```
irb> Pilot.create(name: "Gautam")
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: nil, weight: nil, gforce: nil>
irb> CarDriver.create(name: "Car Gautam")
```

```

=> #<CarDriver _id: 4ef9b206fed0eb9824000001, _type: "CarDriver",
name: "Car Gautam", age: nil, address: nil, weight: nil, license: nil>

irb> ShipDriver.create(name: "Ship Gautam")
=> #<ShipDriver _id: 4ef9b21afed0eb9824000002, _type: "ShipDriver",
name: "Ship Gautam", age: nil, address: nil, weight: nil, can_swim:
nil>

irb> > Marine.count
=> 1

> Marine.first
=> #<ShipDriver _id: 4ef9b21afed0eb9824000002, _type: "ShipDriver",
name: "Ship Gautam", age: nil, address: nil, weight: nil, can_swim:
nil>

> Terrestrial.count
=> 1

> Terrestrial.first
=> #<CarDriver _id: 4ef9b206fed0eb9824000001, _type: "CarDriver",
name: "Car Gautam", age: nil, address: nil, weight: nil, license: nil>

irb> Driver.count
=> 3

```

What just happened?

Using Single Collection Inheritance, we can find out how different types of drivers form different levels of specialization.

Let's create a few objects as follows:

```

irb> Pilot.create(name: "Gautam")
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: nil, weight: nil, gforce: nil>

irb> CarDriver.create(name: "Car Gautam")
=> #<CarDriver _id: 4ef9b206fed0eb9824000001, _type: "CarDriver",
name: "Car Gautam", age: nil, address: nil, weight: nil, license: nil>

irb> ShipDriver.create(name: "Ship Gautam")
=> #<ShipDriver _id: 4ef9b21afed0eb9824000002, _type: "ShipDriver",
name: "Ship Gautam", age: nil, address: nil, weight: nil, can_swim:
nil>

```

Here we created a `Pilot`, `ShipDriver`, and a `CarDriver` object. All in the standard normal way of creating objects. However, we can also access these objects in different ways.

```
> Marine.first  
=> #<ShipDriver _id: 4ef9b21afed0eb9824000002, _type: "ShipDriver",  
name: "Ship Gautam", age: nil, address: nil, weight: nil, can_swim:  
nil>
```

Remember that we never created a `Marine` object. However, when we try to fetch the first `Marine` object, it works! Notice that even the type of object fetched is not a `Marine` but a `ShipDriver` object. What's going on? We wanted to fetch the first `Marine` object and it returned a `ShipDriver` object!

This is polymorphism in action. The `Marine` class behaves in different ways depending on the object it represents. In other words, the `Marine` class has a polymorphic relation with its subclasses.

Going deeper into this:

```
irb> Driver.count  
=> 3
```

We created a `Pilot`, `ShipDriver`, and a `CarDriver` but the `Driver` count is 3.

Basic polymorphic relations

Now let's see a different way of managing polymorphic relations. Let's consider the vehicles. There are different types of vehicles—all having totally different properties but all are vehicles nevertheless. So, SCI may not be a good choice for a space shuttle and a bike, as they are entirely different vehicles!

Choosing SCI or basic polymorphism.

What you need to consider is the number of collections you want. If you want all objects to reside in one collection use SCI. If you want objects to reside in different collections use basic polymorphism.

In other words, in case the polymorphism is data-centric (that is, if objects have a lot of different properties or data), use basic polymorphism.

If the polymorphism is more functionality-centric (that is, if objects have similar properties but different functions) use SCI.



Time for action – creating vehicles using basic polymorphism

Let's design the Vehicle model:

```
# app/models/vehicle.rb

class Vehicle
  include Mongoid::Document

  belongs_to :resource, :polymorphic => true

  field :terrain, type: String
  field :cost, type: Float
  field :weight, type: Float
  field :max_speed, type: Float
end
```

This is the main polymorphic class. We now use this class in other models.



Unlike SCI, each model is independent, but can choose to be a part of Vehicle. It has its own identity and does not inherit from any parent model.



Let's create a few objects. The code to create a Bike model is as follows:

```
# app/models/bike.rb
class Bike
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :gears, type: Integer
  field :has_handle, type: Boolean
  field :cubic_capacity, type: Float
end
```

The code to create a Ship model is as follows:

```
# app/models/ship.rb
class Ship
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :is_military, type: Boolean
```

```
    field :is_cruise, type: Boolean
    field :missile_capable, type: Boolean
    field :anti_aircraft, type: Boolean
    field :number_engines, type: Integer
end
```

The code to create a Submarine model is as follows:

```
# app/models/submarine.rb
class Submarine
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :max_depth, type: Float
  field :is_nuclear, type: Boolean
  field :missile_capable, type: Boolean
end
```

The code to create a SpaceShuttle model is as follows:

```
# app/models/space_shuttle.rb
class SpaceShuttle
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :boosters, type: Integer
  field :launch_location, type: String
end
```

The code to create an Aeroplane model is as follows:

```
# app/models/aeroplane.rb
class Aeroplane
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :seating, type: Integer
  field :max_altitude, type: Integer
  field :wing_span, type: Float
end
```

The code to create a Car model is as follows:

```
# app/models/car.rb
class Car
  include Mongoid::Document

  has_one :vehicle, :as => :resource

  field :windows, type: Integer
  field :seating, type: Integer
  field :bhp, type: Float
end
```

Here, you see that each model has a bunch of properties that are different from each other but all basically fall under the Vehicle category. One of the advantages of basic polymorphism is that it's easy to enter and exit from this pattern. It's very easy to incorporate an existing model into a polymorphic pattern and equally easy to remove an existing model from one. We just add or remove the relationship to the polymorphic model.

Now let's build objects as follows:

```
irb> ship = Ship.new(is_military: true)
=> #<Ship _id: 4f042c53fed0ebc45b000003, _type: "Ship", is_military:
true, is_cruise: nil, missile_capable: nil, anti_aircraft: nil,
number_engines: nil>

irb> vehicle = Vehicle.create(resource: ship)
=> #<Vehicle _id: 4f042c87fed0ebc481000002, _type: "Vehicle",
resource_type: "Ship", resource_id: BSON::ObjectId('4f042c53fed0ebc4
5b000003'), terrain: nil, cost: nil, weight: nil, max_speed: nil>
```

What just happened?

We created a Ship object and then associated it to Vehicle. Let's have a closer look at this in the following code:

```
irb> vehicle = Vehicle.create(resource: ship)
=> #<Vehicle _id: 4f042c87fed0ebc481000002, _type: "Vehicle",
resource_type: "Ship", resource_id: BSON::ObjectId('4f042c53fed0ebc4
5b000003'), terrain: nil, cost: nil, weight: nil, max_speed: nil>
```

Notice the resource_id and resource_type fields, they define the resource that the vehicle represents. To get actual information about the vehicle, we have to lookup the Ship object.

This two-step process could have been done in one step itself, as follows:

```
irb> Vehicle.create(resource: Ship.create(is_military: true))
=> #<Vehicle _id: 4f042de8fed0ebc4c5000004, _type: "Vehicle",
resource_type: "Ship", resource_id: BSON::ObjectId('4f042de8fed0ebc
4c5000003'), terrain: nil, cost: nil, weight: nil, max_speed: nil>
```

Remember, that we cannot do this the other way round:

```
irb> ship = Ship.create(:vehicle => Vehicle.create)
=> #<Ship _id: 4f042dd0fed0ebc4c5000002, _type: "Ship", is_military:
nil, is_cruise: nil, missile_capable: nil, anti_aircraft: nil, number_
engines: nil>

irb> Vehicle.last
=> #<Vehicle _id: 4f042dd0fed0ebc4c5000001, _type: "Vehicle",
resource_type: nil, resource_id: nil, terrain: nil, cost: nil, weight:
nil, max_speed: nil>

irb> Vehicle.create(:resource => Ship.create)
```

When the first command is run, the `Vehicle` object is created first, so the `Ship` object cannot be assigned as the resource. That is the reason the `Vehicle` object has `resource_type` and `resource_id` as `nil`. Obvious, wasn't it?

Choosing SCI or basic polymorphism

As mentioned earlier, this is the choice of single collection or multiple collections. It's best shown by an example. The MongoDB collection looks like the following for drivers and vehicles:

```
> db.drivers.find()
{"_id":ObjectId("..."), "name":"Gautam", "_type":"Pilot" }
{"_id":ObjectId("..."), "name":"Gautam", "_type":"CarDriver" }
{"_id":ObjectId("..."), "name":"Gautam", "_type":"ShipDriver" }
```

Notice, that for the `drivers` collection, the `_type` of objects are different in the same collection. This is SCI!

```
> db.vehicles.find()
{"_id":ObjectId("..."), "_type" : "Vehicle", "resource_id" : ObjectId("4f
02077dfed0ebb308000001"), "resource_type" : "Ship" }
{"_id":ObjectId("..."), "_type" : "Vehicle", "resource_id" : ObjectId("4f
020807fed0ebb308000007"), "resource_type" : "Ship" }
```

However, in the `vehicles` collection, the `_type` of objects is the same—`Vehicle`. This is basic polymorphism.

Using embedded objects

We know what embedded objects are and we have seen this already in the previous chapters. Now, we shall see how these are built via DataMappers. Just to recap, an embedded document is one that resides inside a parent document. We have seen a sample of this already, it's listed next:

```
book : { name: "Oliver Twist",
        ...
        reviews: [
        {
            _id: ObjectId("5e85b612fed0eb0bee000001"),
            user_id: ObjectId("8d83b612fed0eb0bee000702"),
            book_id: ObjectId("4e81b95ffed0eb0c23000002"),
            comment: "Very interesting read"
        },
        {
            _id: ObjectId("4585b612fed0eb0bee000003"),
            user_id : ObjectId("ab93b612fed0eb0bee000883"),
            book_id: ObjectId("4e81b95ffed0eb0c23000002"),
            comment: "Who is Oliver Twist?"
        }
    ]
    ...
}
```

In the preceding code, `reviews` is an array of embedded objects. How do you identify an embedded object?

```
{
    _id: ObjectId("5e85b612fed0eb0bee000001"),
    user_id: ObjectId("8d83b612fed0eb0bee000702"),
    book_id: ObjectId("4e81b95ffed0eb0c23000002"),
    comment: "Very interesting read"
}
```

When `ObjectId` exists, it's an embedded object. Now, let's see how we define them using DataMappers. As with all associations, these are two-way associations.

Time for action – creating embedded objects

Let's continue our example and assume that a driver has one address and many bank accounts. As addresses or bank accounts have hardly any relevance without a driver, we choose to embed them into the Driver model.

Using MongoMapper

First let's revisit the Driver model as shown next:

```
class Driver
  include MongoMapper::Document

  one :address
  many :bank_accounts
end
```

Now let's see how the Address and BankAccount models are constructed. This is done as follows:

```
# app/models/address.rb
class Address
  include MongoMapper::EmbeddedDocument

  key :street, String
  key :city, String
end

# app/models/bank_account.rb
class BankAccount
  include MongoMapper::EmbeddedDocument

  key :account_number, String
  key :balance, Float
end
```

Using Mongoid

Using Mongoid, it looks like the following:

```
class Driver
  include Mongoid::Document

  field :name, type: String
  ...
```

```
    embeds_one :address
    embeds_many :bank_accounts
end
```

And the Address and BankAccount models are written as follows:

```
# app/models/address.rb
class Address
  include Mongoid::Document

  field :street, type: String
  field :city, type: String

  embedded_in :driver
end

# app/model/bank_account.rb
class BankAccount
  include Mongoid::Document

  field :account_number, type: String
  field :balance, type: Float

  embedded_in :driver
end
```

If we try this on the Rails console, we can create Driver, Address, and BankAccount objects. Using either of the DataMappers, we can create the objects as follows:

```
irb> d = Driver.first
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name: "Gautam", age: nil, address: nil, weight: nil, gforce: nil>

irb> d.address = Address.new(street: "SB Road", city: "Pune")
=> #<Address _id: 4f0491bcfed0ebcc59000001, _type: nil, street: "SB Road", city: "Pune">

irb> d.bank_accounts << BankAccount.new(account_number: "1230001231225", balance: 1231.23)
=> [#<BankAccount _id: 4f0491f6fed0ebcc59000002, _type: nil, account_number: "1230001231225", balance: 1231.23>]

irb> d.save
=> true

irb> d = Driver.first
```

```
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:  
"Gautam", age: nil, address: {"street"=>"SB Road", "city"=>"Pune", "_  
id"=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:  
nil>  
  
irb> d.address  
=> #<Address _id: 4f0491bcfed0ebcc59000001, _type: nil, street: "SB  
Road", city: "Pune">  
  
irb> d.bank_accounts  
=> [#<BankAccount _id: 4f0491f6fed0ebcc59000002, _type: nil, account_  
number: "1230001231225", balance: 1231.23>]
```

What just happened?

When we add an `Address` object or a `BankAccount` object to `Driver`, an object is created but it's embedded inside the `Driver` object. If we see the MongoDB document, we will notice the following:

```
mongo> db.drivers.findOne()  
{ "_id" : ObjectId("4ef9a410fed0eb977d000002"), "_type" : "Pilot",  
"address" : { "street" : "SB Road", "city" : "Pune", "_id" : ObjectId(  
"4f0491bcfed0ebcc59000001") },  
"name" : "Gautam"  
"bank_accounts" : [  
    {  
        "account_number" : "1230001231225",  
        "balance" : 1231.23,  
        "_id" : ObjectId("4f0491f6fed0ebcc59000002")  
    }  
]
```

Notice that `address` and `bank_accounts` are fields in the document but have `ObjectId` specified in them.



Remember that you cannot create or access embedded objects without the parent object context.

If you try to create an embedded object without any context of the document it's embedded in, you will get an error. We'll see this in the following sections.

Using MongoMapper

```
irb> Address.create  
NoMethodError: undefined method 'create' for Address:Class
```

The Address class does not have a `create` method. This is because it is embedded into another object. Let's see if we can find an address (as weird as that sounds).

```
irb> > Address.first  
NoMethodError: undefined method 'first' for Address:Class
```

That didn't work either—and rightly so.

Using Mongoid

Mongoid gives slightly different errors instead of MongoMapper:

```
irb> Address.create  
NoMethodError: undefined method 'new?' for nil:NilClass
```

Undefined method!! That's a weird one! If we dig deeper into the Mongoid code, we see that a model maps to a collection and we create documents inside that collection. Address is not a collection (as it's an embedded document). So, when we call `create` on this, it tries to resolve that model to collection. As there is no collection by this name, `nil` is passed to the Persistence module, resulting in the `NilClass` error. Not very intuitive, but please pardon Mongoid!

```
irb> Address.first  
Mongoid::Errors::InvalidCollection: Access to the collection for  
Address is not allowed since it is an embedded document, please access  
a collection from the root document.
```

Wow! Finally we get an error that makes sense. Mongoid tells us to access the parent document and not access the embedded document, as there is no collection named `Address`.



This error also gives more insight into how different the internal behavior of Mongoid and MongoMapper is.



Reverse embedded relations in Mongoid

The reverse embedded relations for embedded documents is very important. Mongoid uses them to resolve where these documents are to be embedded. Here are some things we should keep in mind to avoid unforeseen behavior.

Time for action – using embeds_one without specifying embedded_in

If we only specify the `embeds_one` relationship in the parent but do not specify the `embedded_in` relationship in the embedded relation, the document will not be embedded and there will be no error issued either. Have a look at the following code:

```
class Driver
  include Mongoid::Document

  ...
  embeds_one :address
end

class Address
  include Mongoid::Document

  # have intentionally not put the embedded_in relation.
end
```

If we now try to embed the `Address` object into the `Driver`, a half-baked `Driver` object gets created:

```
irb> d = Driver.first
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: {"street"=>"SB Road", "city"=>"Pune", "_id"=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:
nil>

irb> d.address = Address.new(street: "A new street")
=> #<Address _id: 4f0662c2fed0ebe0ee000002, _type: nil, street: "A
new street", city: nil>

irb> d.save
=> true

irb> Driver.first
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: {"street"=>"SB Road", "city"=>"Pune", "_id"=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:
nil>
```

What just happened?

Notice that the address has not changed in the object saved to database, even though MongoDB says that the object was saved correctly. The reason why the address did not change from SB Road to A new street is because when Mongoid tried to save the embedded document, it looked for the reverse relation and did not find it, so that data was ignored.

Under the cover, Mongoid treats embedded models also as `Mongoid::Document`. The `embedded_in` method helps resolve the parent.

Time for action – using `embeds_many` without specifying `embedded_in`

Not specifying the `embedded_in` can cause some real problems even for a many-to-many relation. This would create new half-baked parent objects in the collection. Have a look at the following code:

```
class Driver
  include Mongoid::Document

  ...
  embeds_many :bank_accounts
end

class BankAccount
  include Mongoid::Document

  # have intentionally not put the embedded_in relation.
end
```

Now, if we try to add `BankAccounts` to the `Driver` object, we get into trouble! This is shown next:

```
irb> d = Driver.last
=> #<Driver _id: 4f06667cfed0ebe13e000001, _type: nil, name:
nil, age: nil, address: {"_id":>BSON::ObjectId('4f066684fed0ebe1
3e000002')}, weight: nil>

irb> d.bank_accounts << BankAccount.new
=> [#<BankAccount _id: 4f06672cfed0ebe164000001, _type: nil, account_
number: nil, balance: nil>]

irb> Driver.last
=> #<Driver _id: 4f06672cfed0ebe164000001, _type: nil, name: nil,
age: nil, address: nil, weight: nil>
```

What just happened?

First we fetched the last Driver object as follows:

```
irb> d = Driver.last
=> #<Driver _id: 4f06667cfed0ebe13e000001, _type: nil, name:
nil, age: nil, address: {"_id"=>BSON::ObjectId('4f066684fed0ebe1
3e000002')}, weight: nil>
```

Here, we can see that it's a proper Driver object with some addresses embedded in it. We also see that the Driver object has the ID 4f06667cfed0ebe13e000001.

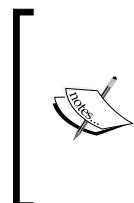
Now, we are trying to embed a BankAccount object into the Driver bank_accounts array but remember that we have not specified the `embedded_in` relation. This is done as follows:

```
irb> d.bank_accounts << BankAccount.new
=> [#<BankAccount _id: 4f06672cfed0ebe164000001, _type: nil, account_
number: nil, balance: nil>]
```

Notice, that we rightly see the BankAccount object inserted into the bank_accounts array. However, there is something seriously wrong in the database update:

```
irb> Driver.last
=> #<Driver _id: 4f06672cfed0ebe164000001, _type: nil, name: nil,
age: nil, address: nil, weight: nil>
```

Now, if we try to fetch the last driver object, we see a Driver object with the ID 4f06672cfed0ebe164000001. This is the object ID of the BankAccount object we created in the earlier step. So, we have a half-baked Driver object.



Be careful! As MongoDB is a schema-free database, it will allow such incorrect behavior to creep in—but it's only we who are to blame when we use Mongoid incorrectly.
MongoMapper, on the other hand, treats embedded documents differently as they are MongoMapper::EmbeddedDocuments, so this problem does not arise.

Understanding embedded polymorphism

Yes! We can use polymorphism even for embedded documents. Why treat them differently? We already know the concept of polymorphism. Let's extend this to embedded documents too.

Single Collection Inheritance

Let's assume that a driver has different types of licenses—to fly, to drive a car, to drive a bike, to drive a ship, to command a space shuttle, among others. As the license cannot exist without a driver, we embed it into the `Driver` model. However, the license shows polymorphic behavior.

Time for action – adding licenses to drivers

First, let's embed licenses into the `Driver` model using Single Collection Inheritance. This can be done as follows:

```
class Driver
  include Mongoid::Document

  field :name, type: String
  ...

  embeds_many :licenses
end
```

And now let's create a `License` model as follows:

```
# app/models/license.rb
class License
  include Mongoid::Document

  embedded_in :driver
end

# app/models/car_license.rb
class CarLicense < License
end
```

Let's see how to embed the `License` model into the `Driver` model in the following code:

```
irb> d = Driver.first
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: {"street"=>"SB Road", "city"=>"Pune", "_id"=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:
nil>

irb> d.licenses << CarLicense.new
=> [#<CarLicense _id: 4f065ed4fed0ebd605000003, _type: "CarLicense">]

irb> d.save
=> true

irb> Driver.first.licenses
=> [#<CarLicense _id: 4f065ed4fed0ebd605000003, _type: "CarLicense">]
```

What just happened?

We can see that the `licenses` array now has a `CarLicense` object in it. It's also interesting to see from the MongoDB console that the ID was really embedded:

```
{ "_id" : ObjectId("4ef9a410fed0eb977d000002") , "_type" : "Pilot" ,  
"address" : { "street" : "SB Road" , "city" : "Pune" , "_id" : ObjectId(  
"4f0491bcfed0ebcc59000001") } , "bank_accounts" : [  
    {  
        "account_number" : "1230001231225" ,  
        "balance" : 1231.23 ,  
        "_id" : ObjectId("4f0491f6fed0ebcc59000002")  
    }  
] , "licenses" : [  
    {  
        "_id" : ObjectId("4f065ed4fed0ebd605000003") ,  
        "_type" : "CarLicense"  
    }  
] , "name" : "Gautam" }
```

Yes it was indeed!

Basic embedded polymorphism

Let's consider the case of insurance for drivers. Assume that drivers may or may not have insurance. For example, suppose we say that pilots and astronauts must have travel insurance and car drivers must have theft insurance. Bike riders don't need any insurance. In such a case, we don't want insurance to be a part of the `Driver` model.

Instead, we should have the option to put it in any class that really needs it. This also means that these insurance classes may be related to different driver subclasses. As insurance is moot without the driver's existence, we should embed it.

Time for action – insuring drivers

Let's prepare different types of insurance as follows:

```
# app/models/pilot.rb  
class Pilot < AeroSpace  
    embeds_many :insurances, as: :insurable  
end  
  
# app/models/car_driver.rb  
class CarDriver < Terrestrial  
    embeds_many :insurance, as: :insurable
```

```
end

# app/models/astronaut.rb
class Astronaut < AeroSpace
  embeds_many :insurances, as: :insurable
end
```

And now we design the Insurance class as follows:

```
# app/models/insurance.rb
class Insurance
  include Mongoid::Document

  embedded_in :insurable, polymorphic: true
end

# app/models/travel_insurance.rb
class TravelInsurance < Insurance
end

# app/models/theft_insurance.rb
class TheftInsurance < Insurance
end
```

Now let's provide insurance policies for our drivers as follows:

```
irb> p = Pilot.first
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:
"Gautam", age: nil, address: {"street"=>"asfds", "city"=>"Pune", "_id":
=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:
nil>

irb> p.insurances << TravelInsurance.new
=> [#<TravelInsurance _id: 4f06ad2efed0ebe598000002, _type:
"TravelInsurance">]

irb> a = Astronaut.first
=> #<Astronaut _id: 4f069fd8fed0ebe45d000001, _type: "Astronaut",
name: nil, age: nil, address: nil, weight: nil, gforce: nil>

irb> a.insurances << TravelInsurance.new
=> [#<TravelInsurance _id: 4f06b058fed0ebe598000004, _type:
"TravelInsurance">]

irb> a.insurances << FireInsurance.new
```

```
=> [#<FireInsurance _id: 4f06ad6bfed0ebe598000003, _type:  
"FireInsurance">]  
  
irb> a.insurances  
=> [#<FireInsurance _id: 4f06ad6bfed0ebe598000003, _type:  
"FireInsurance">, #<TravelInsurance _id: 4f06b058fed0ebe598000004,  
_type: "TravelInsurance">]
```

What just happened?

Let's have a closer look at the preceding commands:

```
irb> p = Pilot.first  
=> #<Pilot _id: 4ef9a410fed0eb977d000002, _type: "Pilot", name:  
"Gautam", age: nil, address: {"street"=>"asfds", "city"=>"Pune", "_id":  
"=>BSON::ObjectId('4f0491bcfed0ebcc59000001')}, weight: nil, gforce:  
nil>  
  
irb> p.insurances << TravelInsurance.new  
=> [#<TravelInsurance _id: 4f06ad2efed0ebe598000002, _type:  
"TravelInsurance">]
```

Here, `Insurance` is polymorphic. This means that the `Insurance` object can be embedded in multiple parents. In this case, we have `TravelInsurance` (that is, a model, which inherits from `Insurance`) being assigned to the `Pilot` class:

```
irb> a = Astronaut.first  
=> #<Astronaut _id: 4f069fd8fed0ebe45d000001, _type: "Astronaut",  
name: nil, age: nil, address: nil, weight: nil, gforce: nil>  
  
irb> a.insurances << TravelInsurance.new  
=> [#<TravelInsurance _id: 4f06b058fed0ebe598000004, _type:  
"TravelInsurance">]
```

Now, we have the `TravelInsurance` object being embedded in the `Astronaut` class. This shows us the polymorphic nature of the `Insurance` embedded object – it can be embedded in different parents.

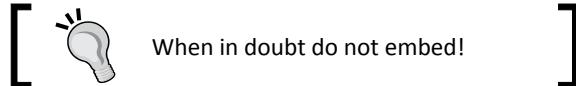
Have a go hero

Why don't you try and assign `TheftInsurance` to `CarDriver`?

Choosing whether to embed or to associate documents

This is indeed sometimes a dilemma. While modeling data, if you see that the child document cannot exist without the parent object and if you are relatively sure that you would not need to search for the child objects directly, you could embed them.

For the UML savvy, a composition relation is a good candidate for embedding.



So, what happens if you embed an object and realize later that you need to process embedded objects? Or maybe the relation was wrong—it should not have been embedded? Don't worry! The following are a couple of options you have:

- ◆ Change the code from embed to association. As MongoDB is schema free, new objects will automatically pick up the relation.
- ◆ Fire queries on the embedded objects if required. But, this may not be a good solution as it would mean unnecessary calls for even basic lookups.

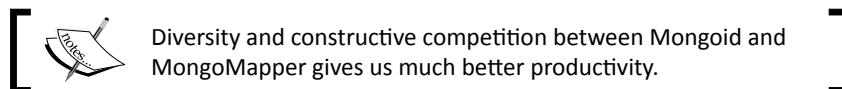
Mongoid or MongoMapper – the verdict

It's neutral! Stick to either Mongoid or MongoMapper, not both at the same time. My personal preference is Mongoid as it's closer to the `ActiveModel` relations than MongoMapper.

The following are some points to ponder:

- ◆ MongoMapper has lesser documentation than Mongoid and it's sometimes not up-to-date.
- ◆ Many-to-many associations are updated only one-sided in MongoMapper. Mongoid gets this right and both objects keep an array of each other, so we can query both ways.
- ◆ Sometime errors spewed by MongoMapper and Mongoid can be intimidating. It usually means we are doing something wrong.
- ◆ There are no embedded reverse associations in MongoMapper. This is advantageous because unlike Mongoid, MongoMapper does not use the reverse association for creating embedded objects. Having it, however, gives better visibility to us and is also more aligned with the `ActiveModel` relations.

Overall, it's a matter of choice. I have chosen Mongoid as my DataMapper. It's also interesting to realize that merging the two into a new MongoDB mapper would be very complex, as both of them work in different ways internally.



Pop quiz – Mongoid, MongoMapper, and more

1. Which of the following does not define a MongoDB aware model?
 - a. include Mongoid::Document
 - b. include MongoMapper::Document
 - c. include MongoMapper::EmbeddedDocument
 - d. include Mongoid::EmbeddedDocument
2. In Mongoid, what is the reverse embedded relation method?
 - a. belongs_to
 - b. embedded_in
 - c. has_many
 - d. has_and_belongs_to_many
3. Which of the following is not true for Single Collection Inheritance?
 - a. All documents are stored in a single collection.
 - b. A single collection contains different types of documents.
 - c. The `resource_id` and `resource_type` determine the document type.
 - d. All models are inherited from a single base model.
4. Which of the following mentions a true difference between Mongoid and MongoMapper?
 - a. Unlike Mongoid, MongoMapper has only one-way association for many-to-many relations.
 - b. Unlike MongoMapper, Mongoid supports embedded polymorphic relations.
 - c. Mongoid has modules and MongoMapper has plugins.
 - d. MongoMapper has Plucky and Mongoid has Criteria.

Summary

In this chapter we learned how MongoDB mappers work using Mongoid and MongoMapper. We saw how we can configure Mongoid and MongoMapper. We then fired queries to fetch, create, and update documents. We also implemented the basic relations—one-to-one, one-to-many, and many-to-many. We played around the concept of polymorphic relations and how we can implement them in documents, as well as embedded documents.

In the next chapter we shall see how we can create a web application using all that we have learned in this chapter. We shall integrate Ruby DataMappers with Rails and Sinatra. If the going was a breeze until now, it gets windy after this!

6

Modeling Ruby with Mongoid

I have been unfair with you in the previous chapters! We have been seeing a lot of Ruby code using MongoMapper and Mongoid but I have not explained how that works. Chapter 4, Working Out Your Way with Queries taught us how to query in MongoDB. Chapter 5, Ruby DataMappers: Ruby and MongoDB Go Hand in Hand showed us how to interact with MongoDB from Ruby. In this chapter, we once again change gears and shall look at the first step to get our Ruby application onto the web, building models using Mongoid. This is one step closer to the web application we want to build!

In this chapter we shall learn the following:

- ◆ Setting up a Mongoid project in Rails, Sinatra, and a simple Rack application
- ◆ Defining attributes in Mongoid and their options
- ◆ Defining different types of relations in Mongoid
- ◆ Using arrays and hashes in our model
- ◆ Embedding documents in the model
- ◆ Setting up indexes for faster querying
- ◆ Making changes in our models and the impact it has on the database documents

Developing a web application with Mongoid

Choices are tough but inevitable—Mongoid or MongoMapper? This book here onwards would use Mongoid as its data mapper and we shall see more of web development using Ruby and MongoDB via Mongoid.

Setting up Rails

We have already seen in the earlier chapter how to set up a Rails application for Mongoid and MongoMapper. Here is a summary again.

Time for action – setting up a Rails project

We are using Rails 3 to set up a new project and we shall continue our library management system: Sodibee. We can set up Rails for Sodibee using the following commands:

```
$ rails new sodibee -OT  
  create  
  create  README  
  create  Rakefile  
...  
  create  vendor/plugins/.gitkeep  
  run    bundle install  
$
```

Now, verify that the config/application.rb has the following code in it. Notice that the ActiveRecord railtie is commented out:

```
require File.expand_path('../boot', __FILE__)  
  
# Pick the frameworks you want:  
# require "active_record/railtie"  
require "action_controller/railtie"  
require "action_mailer/railtie"  
require "active_resource/railtie"  
require "rails/test_unit/railtie"
```

 A **railtie** is a class that sits at the core of the Rails framework. It's the glue that ties in every component into the Rails core framework. Using railties, we can easily add/modify the Rails initialization process and add/extend the Rails framework.

What just happened?

Let's briefly look at the options we have when initializing a Rails project:

- ◆ -O: Using this option, the Rails project skips Active Record files
- ◆ -T: Using this option, the Rails project skips Test::Unit files.

We can now configure Mongoid into the Rails application. First, ensure that the Gemfile has Mongoid configured:

```
gem 'mongoid'
gem 'bson'
gem 'bson_ext'
```



Ensure that bson, bson_ext, and mongo gems have the same version!
At the time of writing this book, I was using version 1.6.2.

Now ensure that Mongoid is configured properly:

```
$ rails generate mongoid:config
```

This generates the config/mongoid.yml file that has some default configuration for the database connectivity. The file should look like the following:

```
development:
  host: localhost
  database: sodibee_development

test:
  host: localhost
  database: sodibee_test

# set these environment variables on your prod server
production:
  host: <%= ENV['MONGOID_HOST'] %>
  port: <%= ENV['MONGOID_PORT'] %>
  username: <%= ENV['MONGOID_USERNAME'] %>
  password: <%= ENV['MONGOID_PASSWORD'] %>
  database: <%= ENV['MONGOID_DATABASE'] %>
  # slaves:
  #   - host: slave1.local
  #     port: 27018
  #   - host: slave2.local
  #     port: 27019
```

Setting up Sinatra

When using Sinatra remember only two words: light-weight and Rack. We can write a fully functional web application in four lines of code:

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

Sinatra was a rebel that was welcomed. There was a time when ActiveRecord ruled and was so tightly coupled with Ruby on Rails that it was virtually impossible to use anything else. The controllers packed so much in them, that the framework became really heavy.

Blake Mizerany wrote Sinatra as a light-weight framework. It came with minimal or no baggage and ran as a simple Rack application! Merb too made a strong appearance around this time but it was heavier than Sinatra and lighter than Rails (2.x).

The Rails 3 core team realized the value of being pluggable and redesigned the architecture with Metal. **Metal** is a pluggable middleware manager, where one can configure how heavy the framework should be. Today, Rails 3 can do everything as lightly as Sinatra can do and even allows a seamless addition of our own middleware in the Rack – so for the remainder of this book we will see Rails 3!

Kudos to Sinatra and Merb!

The modular version of building a Sinatra application requires only two files primarily—the config.ru and a main application code file. A typical config.ru would look like the following:

```
# This file is used by Rack-based servers to start the application.

require 'sinatra'
require './app'

run Sinatra::Application
```

The app.rb (our application code file) looks like the following:

```
require 'sinatra'

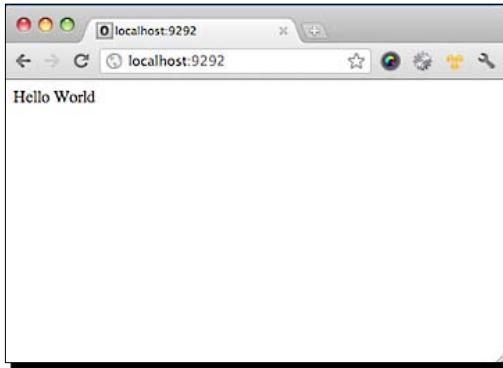
get "/" do
  "Hello Word"
end
```

This is almost similar to writing it in a single file except that config.ru is a rackup file, so we can configure it directly with any Rack application. Running this is as simple as follows:

```
$ rackup config.ru

INFO  WEBrick 1.3.1
INFO  ruby 1.9.2 (2011-07-09) [i386-darwin9.8.0]
INFO  WEBrick::HTTPServer#start: pid=16574 port=9292
```

And now when we start the browser, we can see the output:



Time for action – using Sinatra professionally

Now, let's take a little more professional approach by adding a Gemfile to the application. In the same folder as the other two files, let's add the `Gemfile` with the following contents:

```
source :rubygems  
gem 'sinatra'
```

And now we simply bundle this together and run it:

```
$ bundle install  
...  
$ bundle exec rackup config.ru  
  
INFO  WEBrick 1.3.1  
INFO  ruby 1.9.2 (2011-07-09) [i386-darwin9.8.0]  
INFO  WEBrick::HTTPServer#start: pid=16574 port=9292
```

This is now a full-fledged setup.

Now, let's see how we can add Mongoid to this application. We need to simply add models to the application. In other words, just require these model files. Here are the changes we make to the `Gemfile`:

```
source :rubygems  
  
gem 'sinatra'  
gem 'mongoid'  
gem 'bson'  
gem 'bson_ext'
```

As we have included Mongoid, let's also include the Mongoid models. But first, let's create the models in the `models` directory:

```
$ mkdir models
```

And let's add some models. We can add the `Author`, `Book`, and `Category` models as follows:

```
# models/author.rb
class Author
  include Mongoid::Document

  field :name, type: String

end

# models/book.rb
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

end

# models/category.rb
class Category
  include Mongoid::Document

  field :name, type: String

end
```

Now that we have added the models, we should also include them properly in the `config.ru` and also configure MongoDB. The `config.ru` is configured as:

```
require 'sinatra'
require 'mongoid'

require './app'

Dir["models/*.rb"].each do |file|
  require "./models/#{File.basename(file, '.rb')}"
end

run Sinatra::Application
```

And this is what the code in the main application file, called `app.rb`, should look like:

```
# app.rb

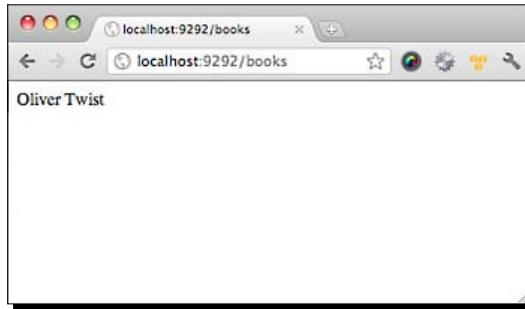
require 'mongoid'
require 'sinatra'

configure do
  Mongoid.configure do |config|
    name = "sodibee_development"
    host = "localhost"
    config.master = Mongo::Connection.new.db(name)
    config.persist_in_safe_mode = false
  end
end

get "/" do
  "Hello World"
end

get "/books" do
  Book.first.name
end
```

That's it! Let's see what the browser has to say now:



What just happened?

We got MongoDB working using a Sinatra application. Let's see the code in detail. The `Gemfile` needs no explanation as it has the gems we require—`sinatra`, `mongoid`, and `bson_ext`. Let's look at the `config.ru` backup file, it looks like this:

```
require 'sinatra'
require 'mongoid'

require './app'
```

```
Dir["models/*.rb"].each do |file|
    require "./models/#{File.basename(file, '.rb')}"
end

run Sinatra::Application
```

Requiring the `mongoid` and `sinatra` gems is straightforward. However, we also need to include `app.rb`—the main application. Let's have a look at the `config.ru` rackup file again:

```
require 'sinatra'
require 'mongoid'

require './app'

Dir["models/*.rb"].each do |file|
    require "./models/#{File.basename(file, '.rb')}"
end

run Sinatra::Application
```

The highlighted code lists all the `.rb` files in a directory and loads them. Let's take a look at `config.ru` a second time:

```
require 'sinatra'
require 'mongoid'

require './app'

Dir["models/*.rb"].each do |file|
    require "./models/#{File.basename(file, '.rb')}"
end

run Sinatra::Application
```

The highlighted code is a call to actually run the Sinatra application. Remember, that we have already loaded the application file that has routes, configuration, and control code!

Let's have a look at the main application file `app.rb`:

```
# app.rb

require 'mongoid'
require 'sinatra'

configure do
```

```
Mongoid.configure do |config|
  name = "sodibee_development"
  host = "localhost"
  config.master = Mongo::Connection.new.db(name)
  config.persist_in_safe_mode = false
end

get "/" do
  "Hello World"
end

get "/books" do
  Book.first.name
end
```

The configure block sets up MongoDB. We set the name as well as host and use the mongo-ruby-driver to configure the database. Now, all the models that have `mongoid` included in them and they can directly access the database!

Have a look at `app.rb` again:

```
# app.rb

require 'mongoid'
require 'sinatra'

configure do
  Mongoid.configure do |config|
    name = "sodibee_development"
    host = "localhost"
    config.master = Mongo::Connection.new.db(name)
    config.persist_in_safe_mode = false
  end
end

get "/" do
  "Hello World"
end

get "/books" do
  Book.first.name
end
```

This is the web server root path. That means that if the URL does not contain anything but the domain and the port, this path will be used. An application must have at least this route defined to work.

Let's take a look at `app.rb` a third time:

```
# app.rb

require 'mongoid'
require 'sinatra'

configure do
  Mongoid.configure do |config|
    name = "sodibee_development"
    host = "localhost"
    config.master = Mongo::Connection.new.db(name)
    config.persist_in_safe_mode = false
  end
end

get "/" do
  "Hello World"
end

get "/books" do
  Book.first.name
end
```

Using the `"/books"` route for the Sinatra application, we can directly access the books using the `Book` model. The preceding code prints the name of the first book!



It's interesting to note that the models (`Book`, `Author`, among others) have not changed, whether it's Sinatra or a Rails application!



Understanding Rack

We have heard the word Rack earlier. But what is Rack and what does it mean?

Rack is the glue that binds web frameworks with the web servers. Every web server is expected to respond to HTTP requests with a status, header, and body. Rack simplifies this and defines the standard in which a web server should respond. The simplest Rack application is:

```
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
end
```

The previous code is from one of the famous resources for introducing Rack <http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html>. This is an excellent example to understand what Rack means. In the preceding code, 200 represents the HTTP status code, { "Content-Type" => "text/plain" } represents the HTTP headers, and [" Hello world! "] is the HTTP body.

Simple and sweet! Where do Sinatra and Rails fit in? They fit right into the Rack by implementing the call method internally.

Defining attributes in models

Until now we have seen how attributes are added in models. But we never really dug deeper to find out how that works.

A typical model looks like the following:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date

  field :votes, type: Array
  field :reviews, type: Hash
end
```

The field method from Mongoid::Document takes at least one mandatory parameter and some optional arguments—name is mandatory and here are some optional arguments. The ones we would use most are :type and :default. The optional arguments are explained as follows:

- ◆ :type: It is the data type which should either be a String, Data, Integer, Float, Bignum, Boolean, or something similar
- ◆ :as: This is required when specifying a polymorphic relation
- ◆ :default: This sets the default value to the field
- ◆ :localize: It tells Mongoid that this is i18n compliant
- ◆ :identity: This is for specifying the information for the identity map

We may not specify any options. This is taken as an on-the-fly configuration. It's advantageous if all the fields are strings or we know what we are typecasting them as. This improves performance but is not recommended. It also leads to code readability issues and could cause problems later.

The `:default` option is very interesting. It can be set to a value or even be a block of code:

```
field :published_on, default: Time.now
```

Alternatively, we could also use a block of code for default:

```
field :published_on, default: { Time.now - 2.years }
```

Accessing attributes

We access the attributes in any of the following ways:

- ◆ `book = Book.first`
- ◆ `book.name` # => "Oliver Twist"
- ◆ `book[:name]` # => "Oliver Twist"
- ◆ `book.read_attribute(:name)` # => "Oliver Twist"

Similarly, we can set values too, as follows:

- ◆ `book.name = "Something Else"`
- ◆ `book[:name] = "Something Else"`
- ◆ `book.write_attribute(:name, "Something Else")`

We can also set multiple attributes at the same time, as follows:

```
Book.write_attributes(name: "Something Else", publisher: "Dover")
```

Indexing attributes

Indexing fields improves performance for lookups. We can add various types of indexes to models. Basic indexing is done as follows:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  ...

  index :publisher
end
```

But we can specify different types of indexes too.

Unique indexes

This is the most common type of indexing scheme. We can ensure that the indexes are unique. It is done as follows:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  ...

  index :publisher, unique: true
end
```

Background indexing

Creating indexes in real time can be expensive as it blocks the database operations while creating indexes. Adding the `background` option does indexing in the background, as follows:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  ...

  index :publisher, unique: true, background: true
end
```

Geospatial indexing

We shall see details of geospatial indexing in later chapters. In a nut shell though, when we require a latitude and longitude field for a model, we can leverage the in-built geospatial indexing provided by MongoDB with help from a custom class in `app/models/` named as `location.rb`:

```
class Location
  include Mongoid::Document

  field :coordinates, type: Array

  index [ [:coordinates, Mongo::GEO2D] ]
end
```

Sparse indexing

When we don't want to index every document but only those that have any indexed fields, we term it as a sparse index. It's done as follows:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  ...

  index :publisher, sparse: true
end
```

Remember, that when we use sparse indexes, results returned from the query could be only from the indexed document and not on all the documents in the collection. So, be careful.



Currently, there can be only one indexed field as a sparse.



Dynamic fields

As MongoDB is schema free, does it mean that we can actually define fields on-the-fly? Yes! So, not only do we not need a structured schema, in fact we may not require a schema at all!

This helps in cases where the schema is subject to change frequently. Dynamic fields are turned on by default in Mongoid. This means that if we define a field that does not exist in the schema, it will automagically get added to the document. Isn't that really cool. Let's consider the basic Book model:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  field :name, type: String
end
```

Time for action – adding dynamic fields

Let's see how this works! Execute the following:

```
irb>b = Book.first
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, publisher: "Dover
Publications", name: "Oliver Twist">

irb> b[:dedication] = "The kids"
```

```
=> "The kids"

irb> b.save!
=> true

irb> b
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, publisher: "Dover
Publications", name: "Oliver Twist", dedication: "The kids">
```

What just happened?

As per the Book model, there are only two fields: `publisher` and `name` for a book. However, we can easily add a new field `dedication` to this document. Though it seems straightforward, there are a couple of things that we should know.

For dynamic fields, we do not have the getter/setter routines. It means, for the case just discussed, when we add a dynamic field `dedication` to the document, we cannot access the object with `b.dedication`. That will throw a `NoMethodError` exception as follows:

```
> b.dedication
NoMethodError: undefined method 'dedication' for #<Book:0x1e0e2e0>
...
> b.dedication = "Not for the kids"
NoMethodError: undefined method 'dedication=' for #<Book:0x1e0e2e0>
...
```

Why is it like this, you ask? Well, let's look at it objectively. If, for every dynamic field, the Ruby DataMapper adds a getter/setter routine (that is, `dedication` and `dedication=` methods), the class code will become huge and unmanageable. More importantly, if we add fields whose names conflict with internal method names, it can cause a lot of trouble. So, dynamic fields are only accessible by the `[]` methods that is, `b[:dedication]`.

Localization

Most databases require Localization and Internationalization. In turn, Mongoid and MongoMapper both use the `i18n` gem for internationalization.

Internationalization and Localization are very commonly misunderstood.

Internationalization deals with the process of setting up localization! For example, managing different character encoding schemes (UTF8, UTF16, among others), date formats, currency formats, and so on.

Localization is displaying information based on the locale – language symbols, currency, character markups like é or symbols like a or currency like €, and so on.

Time for action – localizing fields

Let's see how we can configure localized data in Mongoid:

```
class Book
  include Mongoid::Document

  field :publisher, type: String
  field :price, localize: true
end
```

Note, that we have not defined the `type` for the field `price`; instead we have set the `localize` option. This internally tells Mongoid to store this data as a hash! Depending on the different locales supported, the different currency will get set. Let's execute the following commands:

```
irb> b = Book.first
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, publisher: "Dover
Publications", name: "Oliver Twist", price: nil>

irb> I18n.locale
=> :en

irb> b.price = "40$"
=> "40$"

irb> I18n.locale => :hi
=> :hi

irb> b.price = "Rs. 2000"
```

```
=> "Rs. 2000"

irb> b.save
=> true

irb> b
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, publisher: "Dover Publications", name: "Oliver Twist", price: {"en"=>"40$", "hi"=>"Rs. 2000"}>

irb> b.price_translations
=> {"en"=>"40$", "hi"=>"Rs. 2000"}
```

What just happened?

As `price` is defined as a localized field, Mongoid automatically maintained a hash of locales and its localized values. Now, depending on the locale, the information will be displayed:

```
irb> I18n.locale = :en
=> :en

irb> b.price
=> "40$"
```

As we can see, if the locale is `:en`, the price is shown as "`40$`". Similarly, if the locale is `:hi`, the price is shown as "`Rs. 2000`".

```
irb> I18n.locale = :hi
=> :hi

irb> b.price
=> "Rs. 2000"
```



Ensure that you have a Mongoid version greater than 2.4.0!



Using arrays and hashes in models

Just like we have fields with different basic data types, we can also add fields as arrays and hashes. They make the models richer.



Arrays are used for sequential storage. Hashes are used for quicker lookups.
This acts as the basis for choosing an array or a hash to store data.



This is how we define them in the models:

```
class Book
  include Mongoid::Document

  field :votes, type: Array
  field :reviews, type: Hash

end
```

Let's add some votes to the Book as follows:

```
irb> b = Book.first
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications">

irb> b.votes << [ {"username"=>"Gautam", "rating"=>3} ]
=> [{"username"=>"Gautam", "rating"=>3} ]

irb> vote = b.votes[0]
=> {"username"=>"Gautam", "rating"=>3}

irb> vote['username']
=> 'Gautam'
```

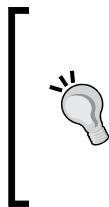
Now let's add some reviews to a book, as follows:

```
irb> b.reviews["Gautam"] = "Very entertaining book"
=> "Very entertaining book"

irb> b
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", vote: [{"username"=>"Gautam",
"rating"=>3}], reviews: { "Gautam" => "Very entertaining book" }>
```

Embedded objects

We can embed documents using relations, as we shall see later on in this chapter. Embedded documents look like hashes with keys and values with the exception that they have the `_id` field as the object ID.



When should we embed objects and when should we just use hashes?

`ActiveModel` callbacks are called on embedded objects unlike direct hashes. So, if we need to do some pre-processing (like setting default values to the object) or post-processing (maybe logging in to a remote service or sending e-mail notifications), we can use the `ActiveModel` callbacks like `before_save` and `after_save` in embedded objects.

Defining relations in models

Let's see how relations are set up using Mongoid. We have seen a preview in earlier chapters about this. Now, we shall take a deeper dive. We have taken the top down approach earlier and seen the following:

- ◆ Many-to-one relations
- ◆ One-to-one relations
- ◆ Many-to-many relations
- ◆ Polymorphic relations

Now, we shall see a different side to them. We shall study the different relations based on the method options available. All relations when defined in the models can be configured minutely using different parameters, as follows:

- ◆ `:name`: This is a mandatory name of the relation and is a symbol by which the relation will be referenced
- ◆ `:options`: It is a hash that is used to configure the relation
- ◆ `:block`: This is an optional block of code to configure some relations

Common options for all relations

The following options are common for all the relations:

- ◆ `:class_name`: The class name if it's not determined from the name.
- ◆ `:extend`: This is the module which will be extended.
- ◆ `:inverse_class_name`: This is used to determine the foreign key.

- ◆ `:inverse_of`: This is the reverse relation, it is very important for creating or embedding relations.
- ◆ `:name`: The name of the relation.
- ◆ `:relation`: The type of the relation. (`Referenced::One`, `Embedded::In`, among others).
- ◆ `:validate`: True or false. This is true by default as we validate the relation.



Among these options `:extend`, `:inverse_class_name`, `:relation` are mostly for internal use. In case we define a new relationship strategy, it would be used. Of course, we would be better off contributing to the Mongoid gem for approval anyway!

:class_name option

In case the related model cannot be deduced from the name, we would need to specify this option:

```
class Foo
  include Mongoid::Document

  has_many :bar_alias, class_name: "Bar"
end
```

Here when we access the relation `bar_alias`, the `Bar` class and its collection would be accessed.

:inverse_of option

In a many-to-many relation, Mongoid saves the information on both sides of the relation. This is called the inverse relation. We shall see a more detailed example in the many-to-many relation later.

:name option

Suppose we want to reference relations with a different name, then we use this option. For example, if we had location information embedded into different documents, they would need to be referenced by different names. We shall see an example of this soon.

Relation-specific options

Some of the following options are applicable to each relation. As we study the relations, we shall see which ones are applicable to which relation. The following is a summary of what they mean:

- ◆ :as: This option is required when defining polymorphic relations
- ◆ :autosave: This option saves the related child automatically when the parent is saved
- ◆ :dependent: We use this option to destroy all child objects just like a cascaded delete
- ◆ :foreign_key: This option indicates an explicitly defined foreign key
- ◆ :order: Set the default order for the relation
- ◆ :index: This option indicates the indexed relation field
- ◆ :polymorphic: This option specifies if the relation is a polymorphic relation
- ◆ :cyclic: This option specifies if a relation is a cyclic embedded relation.
- ◆ :cascaded_callbacks: This option invokes cascaded callbacks on embedded objects
- ◆ :versioned: This option helps manage versions of embedded documents

We shall see where these relations make sense and also look into their details and study the various relations.

Options for has_one

As the method name suggests, this sets up the parent relation for a model having only one child:

```
class Book
  include Mongoid::Document

  has_one :book_detail
end
```

This implies that "A Book has one BookDetail". This method takes the following options:

:as option

When a relation is a polymorphic relation, we need to use this option:

```
class Ship
  include Mongoid::Document

  has_one :vehicle, as: :resource
end
```

This tells the `has_one` method that the vehicle is a polymorphic relation that can be accessed via the `resource_type` and `resource_id` fields in the vehicles collection.

:autosave option

This option is `true` by default. When the object is created, the related child objects are also created. In case the object is updated, only the parent object is updated.

:dependent option

`:dependent` is used for cascaded deletion. We can specify various values:

- ◆ `:delete` and `:delete_all`: This deletes the relation but does not invoke the `ActiveModel :before_delete` and `:after_delete` callback.
- ◆ `:destroy` and `:destroy_all`: This deletes the relation and also invokes the callbacks.
- ◆ `:nullify` and `:nullify_all`: This is used only for embedded documents. When this is specified, the embedded document reference is set to `nil`.

 :`before_delete` and `:after_delete` are `ActiveModel` callbacks. As the names suggest they are invoked before and after any document is deleted.

:foreign_key option

When the referenced key is different and is not the standard `_id` prefix, we need to specify it like this:

```
class Book
  include Mongoid::Document

  has_one :book_detail, foreign_key: :book_detail_info
end
```

Options for has_many

This method sets the parent relation for many child objects. The `has_many` method takes the following options in addition to `:as`, `:autosave`, `:dependent`, and `:foreign_key`.

:order option

We can specify the `order` in a relation as follows:

```
class Author
  include Mongoid::Document

  has_many :books, order: { title: 1 }
end
```

This will get the books of an author sorted by `title` in ascending order.

Options for belongs_to

This is the child side of the relation. It must be set to complement a `has_one` or a `has_many` relation. This method takes the following options in addition to `:autosave` and `:foreign_key`.

:index option

This option determines if the foreign key is indexed or not. It's recommended that the foreign keys be indexed. The values are set to true or false, as shown in the following code:

```
class Book
  include Mongoid::Document

  has_one :review, index: true
end
```

:polymorphic option

We have already seen polymorphic relations in detail. This option sets the polymorphic resource as follows:

```
class Vehicle
  include Mongoid::Document

  belongs_to :resource, :polymorphic => true
end
```

This is used to complement the `:as` option for the parent relationship!

Options for has_and_belongs_to_many

This is the many-to-many relationship method. A typical class would look like the following:

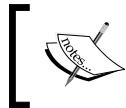
```
class Book
  include Mongoid::Document

  has_and_belongs_to_many :categories
end

class Category
  include Mongoid::Document

  has_and_belongs_to_many :books
end
```

It takes all the standard options such as :autosave, :dependent, :foreign_key, :index, and :order.



A many-to-many relation cannot be a part of a polymorphic relation, as a polymorphic relation expects an explicit parent-child relationship and many-to-many relations are peer relations.



:inverse_of option

Among all the options, the `inverse_of` relation is a very interesting one. As with many-to-many relations, the document IDs are stored as arrays on both sides of the association. So, in the case of `Category` and `Book` objects shown previously, `book_ids` and `category_ids` are arrays that store the `ObjectId` values of the other relations. Let's see the basic many-to-many relation setup. Execute the following commands:

```
irb> b = Book.first
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", name: "Oliver Twist">

irb> c = Category.first
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:
"Fiction">

irb> > c.books << Book.first
=> [BSON::ObjectId('4e86e45efed0eb0be0000010')]

irb> b.categories << c
=> [BSON::ObjectId('4e86e4cbfed0eb0be0000012')]

irb> b
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications", category_ids: [BSON::ObjectId('4e86e4
cbfed0eb0be0000012')], name: "Oliver Twist">

irb> c
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:
"Fiction", book_ids: [BSON::ObjectId('4e86e45efed0eb0be0000010')]>
```

In the following code, we can see that both the related objects, `Book` and `Category`, keep the array `[BSON::ObjectId()]` that contains object ID references of each other:

```
irb> b
=> #<Book _id: 4e86e45efed0eb0be0000010, _type: nil, title: nil,
publisher: "Dover Publications",
category_ids: [BSON::ObjectId('4e86e4cbfed0eb0be0000012')],
```

```

name: "Oliver Twist">

irb> c
=> #<Category _id: 4e86e4cbfed0eb0be0000012, _type: nil, name:
"Fiction",
book_ids: [BSON::ObjectId('4e86e45efed0eb0be0000010')]>

```

Time for action – configuring the many-to-many relation

The `inverse_of` option helps us configure this a little more. If we want only one-sided references to be stored, we can set this flag to false. By default the flag would be true. In this case, if we did not want to store the `category_ids` in the `Book` object, we could change it a little:

```

class Category
  include Mongoid::Document

  has_and_belongs_to_many :books, inverse_of: nil
end

```

Let's see what happens when we execute the following:

```

irb> b = Book.new
=> #<Book _id: 4ef5ab79fed0eb89bf000002, _type: nil, title: nil,
publisher: "Dover Publications", category_ids = [], category_name:
"Oliver Twist">

irb> c = Category.last
=> #<Category _id: 4ef5b48efed0eb8d17000001, _type: nil, name:
"Drama", book_ids: []>

irb> c.books << b
=> [BSON::ObjectId('4ef5ab79fed0eb89bf000002')]

irb> c
=> #<Category _id: 4ef5b48efed0eb8d17000001, _type: nil, name:
"Drama", book_ids: [BSON::ObjectId('4ef5ab79fed0eb89bf000002')]>

irb> b
=> #<Book _id: 4ef5ab79fed0eb89bf000002, _type: nil, title: nil,
publisher: "Dover Publications", category_ids = [], category_name:
"Oliver Twist">

```

What just happened?

Seems almost as similar to the earlier version. However, let's take a closer look:

```
irb> c
=> #<Category _id: 4ef5b48efed0eb8d17000001, _type: nil, name:
"Drama",
book_ids: [BSON::ObjectId('4ef5ab79fed0eb89bf000002')]>

irb> b
=> #<Book _id: 4ef5ab79fed0eb89bf000002, _type: nil, title: nil,
publisher: "Dover Publications",
category_ids = [],
category_name: "Oliver Twist">
```

Notice that the inverse relation was not set in Book object. In other words, as the `inverse_of` was `nil`, the array that should have contained the object IDs of the categories, is empty. In the preceding example `category_ids` will not be updated only if the Category object is updated with books.



If you update the books with categories, that is, `b.categories << c`, then `category_ids` in the Book object will get populated.
I leave it for you to decide if this is a bug or a feature?



Let's see another example in the following section.

Time for action – setting up the following and followers relationship

Let's see if we can set up following and followers between authors. An author can follow other authors and be followed by others too:

```
class Author
  include Mongoid::Document

  has_and_belongs_to_many :followers,
    class_name: "Author",
    inverse_of: :following

  has_and_belongs_to_many :following,
    class_name: "Author",
    inverse_of: :followers

end
```

Let's set up some relationships between authors as follows:

```
irb> > a = Author.first
=> #<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name: "Charles
Dickens", follower_ids: [], following_ids: []
irb> > b = Author.last
=> #<Author _id: 4ef5ab6ffed0eb89bf000001, _type: nil, name: "Mark
Twain", follower_ids: [], following_ids: []
irb> a.following << b
=> [BSON::ObjectId('4ef5ab6ffed0eb89bf000001')]
irb> a
=> #<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name: "Charles
Dickens", follower_ids: [], following_ids: [BSON::ObjectId('4ef5ab6ffe
d0eb89bf000001')]>
irb> b
=> #<Author _id: 4ef5ab6ffed0eb89bf000001, _type: nil, name: "Mark
Twain", follower_ids: [BSON::ObjectId('4e86e4b6fed0eb0be0000011')], 
following_ids: []
irb> a.following
=> [#<Author _id: 4ef5ab6ffed0eb89bf000001, _type: nil, name: "Mark
Twain", follower_ids: [BSON::ObjectId('4e86e4b6fed0eb0be0000011')], 
following_ids: []]
irb> b.followers
=> [#<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name:
"Charles Dickens", follower_ids: [], following_ids: [BSON::ObjectId('4
ef5ab6ffed0eb89bf000001')]]
```

What just happened?

Here, let's analyze the code carefully! We wanted followers and following between authors. As an author can have many followers and can also follow many authors, we set this up as a many-to-many relation. This is shown next:

```
class Author
  include Mongoid::Document

  has_and_belongs_to_many :followers,
    class_name: "Author",
    inverse_of: :following

  has_and_belongs_to_many :following,
    class_name: "Author",
    inverse_of: :followers
end
```

Note that it's the `Author` model that an author follows and can get followed. So the class name is the same. This is also called a recursive relation:

```
class Author
  include Mongoid::Document

  has_and_belongs_to_many :followers,
    class_name: "Author",
    inverse_of: :following

  has_and_belongs_to_many :following,
    class_name: "Author",
    inverse_of: :followers
end
```

Now, we want to maintain different arrays for following and followers. So, whenever we define the follower relation, we need to update its counterpart or the inverse relation too! That is why the `:following` relation has `inverse_of :followers` and vice versa! This is shown clearly in the following code:

```
class Author
  include Mongoid::Document

  has_and_belongs_to_many :followers,
    class_name: "Author",
    inverse_of: :following

  has_and_belongs_to_many :following,
    class_name: "Author",
    inverse_of: :followers
end
```

Now, let's see the actual working of this relationship. When we set up the following for one author, we did it as follows:

```
irb> a.following << b
=> [BSON::ObjectId('4ef5ab6ffed0eb89bf000001')]
```

When this is done, we can see that the `follower_ids` of the `Author` object `a` and the `following_ids` of the `Author` object `b` are updated together! This is shown in the following code:

```
irb> a.following
=> [#<Author _id: 4ef5ab6ffed0eb89bf000001, _type: nil, name: "Mark
Twain",
follower_ids: [BSON::ObjectId('4e86e4b6fed0eb0be0000011')],
```

```

following_ids: []>

irb> b.followers
=> [#<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name:
"Charles Dickens",
follower_ids: [],
following_ids: [BSON::ObjectId('4ef5ab6ffed0eb89bf000001')]>]

```

Options for :embeds_one

This method sets up the parent embedded relation for a single embedded child. As embedded documents can be polymorphic, the :as option is supported. In addition to this, the other supported options are as follows:

:cascade_callbacks option

As embedded documents are part of the parent, their callbacks are not invoked when the parent is saved. We need to explicitly set this option if we want the embedded child document to process callbacks:

```

class Book
  include Mongoid::Document

  embeds_one :book_info, cascade_callbacks: true
end

```

:cyclic

This is used as an option for recursive or cyclic relationships. This method is very specific for embedded documents. This method is useful for setting up a hierarchy of embedded documents—a single parent and multiple embedded child documents. We shall see this being used with the versioning module too a little later.

Time for action – setting up cyclic relations

We have seen how we can configure an author with following and followers using the inverse_of option. Now, let's build the Author and his followers using cyclic relationships! This can be done as follows:

```

class Author
  include Mongoid::Document

  embeds_many :child_authors, class_name: "Author", cyclic: true
  embedded_in :parent_author, class_name: "Author", cyclic: true

end

```

And let's update the objects as follows:

```
irb> a = Author.first
=> #<Author _id: 4e86e4b6fed0eb0be0000011, _type: nil, name: "Charles
Dickens">

irb> a.child_authors << Author.last
=> true

irb> a.child_authors.first.parent_author
=> #<Author _id: 4ef5ab6ffed0eb89bf000001, _type: nil, name: "Mark
Twain">
```

What just happened?

We now embed an array called `child_authors` into the `Author` document and reference the parent using the `parent_author` field.

We can also do the exact same thing we just saw using the following code:

```
class Author
  include Mongoid::Document

  recursively_embeds_many

end
```

Options for embeds_many

This is a method to embed documents. It takes these additional options including the already explained `:as`, `:cascade_callbacks`, `:cyclic`, and `:order`.

:versioned option

We can version different embedded documents. This should not be used directly but via the versioning module. This automatically embeds versions as an embedded document array in the document. We shall learn about this later in the chapter.

Options for embedded_in

This method tells us which object this is embedded in. It's very important that this be configured when we are setting up the embedded relations.



Without `embedded_in` method in the model, the document would not get embedded at all!

```
class Review
  include Mongoid::Document

  embedded_in :book
end
```

This tells Mongoid that the review document is embedded inside the book.

Have a go hero – embedded polymorphic relations

As we must set the `embedded_in` relation between the parent and the child, how do we embed the same document in different objects? Make it polymorphic! We have seen some examples of how to write polymorphic relations for embedded objects in the previous chapter. Go for it!

:name option

What if we want to save the relation twice in the same parent class? For example, in the `Vehicle` model, we want the source and the destination fields but both are `Location` objects. The `name` option specifies in which field the information would be stored. Have a look at the following code:

```
class Vehicle
  include Mongoid::Document

  embeds_one :source, class_name: "Location"
  embeds_one :destination, class_name: "Location"
end

class Location
  include Mongoid::Document

  embedded_in :vehicle, name: :source
  embedded_in :vehicle, name: :destination
end
```

Let's see how this would work. Execute the following code:

```
irb> v = Vehicle.first
=> #<Vehicle _id: 4f042dd0fed0ebc4c5000001, _type: "Vehicle">

irb> v.source = Location.new
=> #<Location _id: 4f214bf7fed0eb863b000001, _type: nil>

irb> v.destination = Location.new
=> #<Location _id: 4f214bfcfed0eb863b000002, _type: nil>
```

This is how we can embed the same object into the document under different names using the `:name` option just explained.

Managing changes in models

What happens if we require some changes to the document schema?

If this were the SQL book, I would have said that we require some way to use statements like `ALTER TABLE`, `ADD COLUMN`, `CHANGE COLUMN`, and so on. You would need some way to maintain the changes and, if required, roll back the changes.

In Rails, this is done using migrations. A sample migration looks like the following:

```
class RemoveNameToUsers < ActiveRecord::Migration
  def self.up
    remove_column :users, :name
  end

  def self.down
    add_column :users, :name, :string
  end
end
```

The `up` method is called when we are setting up the database and the `down` method is called when we want to rollback.

But wait, this is MongoDB, it's a schema-free database, so what should we do? – Nothing!

Time for action – changing models

Let's take a look at the `Book` model:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
end
```

If we have such a model, what does the object look like? Execute the following command to find out:

```
irb> Book.create(publisher: "Dover")
=> #<Book _id: 4f216427fed0eb86ac000001, _type: nil, title: nil,
publisher: "Dover">
```

Now, suppose we wanted to add a few fields to the `Book` model, how do we do that? Change the code! The code would now look like the following:

```
class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date
end
```

What just happened?

Now, let's see what happens when we create a new object as well as access the earlier one we created. Execute the following commands:

```
irb> Book.create(publisher: "Packt", published_on: Date.today)
=> #<Book _id: 4f21660cfed0eb86ac000002, _type: nil, title: nil,
publisher: "Packt", published_on: 2012-01-26 00:00:00 UTC>
```

So far, so good! But what happens to the earlier object created?

```
irb> Book.where(publisher: "Dover").first
=> #<Book _id: 4f216427fed0eb86ac000001, _type: nil, title: nil,
publisher: "Dover", published_on: nil>
```

Notice the `published_on` field that is `nil`!



Always try to avoid removing fields – it can cause undue trouble.

So, go forth and change the models to your heart's content! No worries.

Mixing in Mongoid modules

Mongoid has a very good way to customize or extend the functionality using modules. Not everything is bundled into the default `Mongoid::Document`. They are bundled as modules and can be included into the classes to make them richer.



Ruby modules can be defined as a bunch of methods that can be included or extended. When we include modules, the methods can be accessed as instance methods. When we extend modules, the methods become class methods.

We shall see a few of the modules that are bundled along with Mongoid. There are plenty of gems available and being contributed which are very helpful.

The Paranoia module

This is a module which can be included if we require soft deletion. Documents are not really deleted but marked for deletion. Basically, a field called `deleted_at` gets added to the object.

When the `:delete` or `:destroy` method is called, the timestamp is set for this field. A default scope is added to the model which fetches only those objects which have `deleted_at = null`.

Time for action – getting paranoid

First let's include the `Paranoia` module:

```
class IAmParanoid
  include Mongoid::Document
  include Mongoid::Paranoia

end
```

That's it! Let's see the impact of this module:

```
irb> IAmParanoid.count
=> 0

irb> a = IAmParanoid.create
=> #<IAmParanoid _id: 4f22eca5fed0eb9dfc000001, _type: nil, deleted_at:
nil>

irb> b = IAmParanoid.create
=> #<IAmParanoid _id: 4f22eca9fed0eb9dfc000002, _type: nil, deleted_at:
nil>

irb> IAmParanoid.count
=> 2

irb> > a.remove
=> true

irb> IAmParanoid.count
=> 1
```

```

irb> a = IAmParanoid.deleted.first
=> #<IAmParanoid _id: 4f22eca9fed0eb9dfc000002, _type: nil, deleted_at:
2012-01-27 18:28:13 UTC>

irb> a.restore
=> 2012-01-27 18:28:13 UTC

irb> IAmParanoid.count
=> 2

```

What just happened?

When we added the `Paranoia` module, it added a field called `deleted_at` into the object.

```

irb> a = IAmParanoid.create
=> #<IAmParanoid _id: 4f22eca9fed0eb9dfc000002, _type: nil,
 deleted_at: nil>

```

When we invoke the `remove` method, the `deleted_at` gets updated. Because the `Paranoia` module is included:

- ◆ A field called `deleted_at` is added to the document.
- ◆ A default criteria is added with the condition `where (:deleted_at => nil)`.
- ◆ A scope called `deleted` is added to `where (:deleted_at.ne => nil)`.

Now, when we invoke any finder or criteria methods, we get all objects apart from the ones removed:

```

irb> a.remove
=> true

irb> IAmParanoid.count
=> 1

```

If we want to fetch the deleted objects, we can use the scope `deleted`:

```

irb> IAmParanoid.deleted.first
=> #<IAmParanoid _id: 4f22eca9fed0eb9dfc000002, _type: nil, deleted_
at: 2012-01-27 18:28:13 UTC>

```

To restore the deleted objects, we can simply call `restore`.



To really delete objects permanently from the database, even if we have included the `Paranoia` module, we can call either the `destroy!` or `delete!` methods.

Versioning

If we want to maintain the changes made to the objects, we can include the Versioning module.

This module embeds a `versions` object and maintains the versions for the object. By default, the latest version is returned for the object attributes. However, we can also fetch earlier versions of the object.

Time for action – including a version

Let's go versioning:

```
class Delta
  include Mongoid::Document
  include Mongoid::Versioning

  field :name, type: String
end
```

Let's see it in action:

```
irb> a = Delta.create
=> #<Delta _id: 4f22f748fed0eb9e6e000003, _type: nil, version: 1, name:
nil>

irb> a.name = "First"
=> "First"

irb> a.save
=> true

irb> a
=> #<Delta _id: 4f22f748fed0eb9e6e000003, _type: nil, version: 2, name:
"First">

irb> a.name = "Second"
=> "Second"

irb> a.save
=> true

irb> a
```

```
=> #<Delta _id: 4f22f748fed0eb9e6e000003, _type: nil, version: 3, name: "Second">

irb> a.revise!
=> true

irb> a
=> #<Delta _id: 4f22f748fed0eb9e6e000003, _type: nil, version: 4, name: "Second">
```

What just happened?

When we included the `Versioning` module:

- ◆ A field called `version` gets added to the document with default value `1`
- ◆ A cyclic relation called `versions` gets added

The model is now configured to update the version every time the object is saved. When it's created the first time, notice that the `version` number is set:

```
irb> a
=> #<Delta _id: 4f22f748fed0eb9e6e000003, _type: nil,
version: 1,
name: nil>
```

Every time, the object is saved, the `version` number is incremented and the `versioned` attributes (that is, all the fields in the document) get saved inside the `versions` embedded object's array and the `version` is incremented.

If we want to update the `version` without any changes, we can use the `revise!` method.



Some more fancy stuff with versioning

- If you want to save the document but don't want to version it, use the `versionless` method. This temporarily disables versioning, for example, `object.versionless (&:save)`.
- If you want to see changes made to the object, use the `:previous_changes` method.
- If you want to see the versioned objects, use the `:versions` method.

Notice, that we mentioned cyclic relationship. We saw this earlier in the embedded relations. For versioning, we need exactly one parent and many child documents of the same class embedded in it!

Pop quiz – dancing with Mongoid models

1. Which of the following is the incorrect way of accessing the `title` field of the `Book` model?
 - a. `Book.first.title.`
 - b. `Book.first[:title].`
 - c. `Book.first.read_attribute(:title).`
 - d. `Book.first.get_title.`
2. When a field is localized, how is that field stored in the database?
 - a. As an embedded object.
 - b. As an array.
 - c. As a hash.
 - d. As a comma-separated string.
3. What does the `cascaded_callbacks` option do?
 - a. Enables callback invocation on the embedded object.
 - b. Cascaded deletes the callbacks in children.
 - c. Enables callback invocation for parent object.
 - d. Disables callback invocation on the embedded object.
4. What would `recursively_embeds_many` in the `Author` model not do?
 - a. Add a cyclic `embeds_many` relation for `Author`.
 - b. It creates an array of embedded objects called `child_authors`.
 - c. Add a field called `parent_author` in the `Author` model.
 - d. Adds a field called `author_count` in the `Author` model.
5. Why do we need to specify the `embedded_in` relation in the embedded Model?
 - a. Mongoid needs to index this embedded object.
 - b. All documents are `Mongoid::Document`. This is the only way Mongoid knows that the document is embedded in another document.
 - c. Mongoid needs to store this in the embedded collection.
 - d. When `Mongoid::EmbeddedDocument` is specified, we do not need this relation, otherwise we need it.

Summary

This chapter took us deeper into modeling Ruby classes using Mongoid. We took a deep dive into how we can set attributes, relations, and use different modules available in Mongoid. We are now getting closer to building our web application! We saw how a Sinatra application is set up as well as where the Rack fits in!

Before we get the web application up and running, I believe it's important to understand performance tuning and optimization. The next chapter deals with this. If you live in the fast lane, skip to *Chapter 8, Rack, Sinatra, Rails and MongoDB – Making Use of them All* where we make use of Rack, MongoDB, Rails, and Sinatra to get the web application up and running!

7

Achieving High Performance on Your Ruby Application with MongoDB

Who doesn't care about performance? After all, that's what matters in the end. We could have the best application but if it does not live up to the mark, it's of no use. How does one know if our application is performing well? How does one gauge if we are doing it right? How do we get the best performance out of our application?

In this chapter we shall see the following:

- ◆ How we can configure MongoDB for high performance
- ◆ How we can leverage Ruby to achieve higher performance with MongoDB
- ◆ What we mean by performance of a web application
- ◆ How we can optimize a web application stack

By the end of this chapter, we shall see how our MongoDB server is configured to power a high performance web application. We shall also see the various techniques available in Ruby for achieving higher performance.

Profiling MongoDB

Let's first understand what we mean by profiling!

How do we know if the queries that we are firing in MongoDB are efficient? How can we measure the time taken for queries and find out which are slow-running queries? If we are able to find this information, we can analyze the results and improve our slow-running queries as well as optimize the queries. This is called **profiling**.



Almost all databases, including relational databases provide tools for profiling and logging slow queries. MongoDB is not different.



Time for action – enabling profiling for MongoDB

We can enable profiling from the command line as well as from the mongo console. Let's start it from the command line, as follows:

```
$ sudo mongod run --config /etc/mongodb.conf --rest -vvvv --profile=1
```

This enables the profiling and sets it at level 1.



There are three modes of profiling:

- 0: This indicates profiling is disabled.
- 1: This indicates profiling suited to write only slow operations.
- 2: This indicates profiling suited to write all operations.

Even if profiling is disabled, the slow queries (the ones taking longer than 100 ms by default) get logged to the console!



If you already have a MongoDB service running, we can enable this from the mongo console, too. This can be done as follows:

```
mongo> db.setProfilingLevel(1)
{ "was" : 0, "slowms" : 100, "ok" : 1 }

mongo>
```

To see profiling in action, we can issue the following commands on the mongo console:

```
mongo> db.system.profile.find()
{ "ts" : ISODate("2012-06-08T07:26:43.186Z"), "op" : "query", "ns" :
"sodibee_development.authors", "query" : { "name" : /in/ }, "nscanned" :
609, "nreturned" : 101, "responseLength" : 6613, "millis" : 10,
"client" : "127.0.0.1", "user" : "" }
```

What just happened?

When we enable profiling, the information is logged in to the `db.system.profile` collection. Let's dig deeper. Have a look at the following:

```
mongo> db.setProfilingLevel(1)
{ "was" : 0, "slowms" : 100, "ok" : 1 }

mongo>
```

The `slowms` option tells MongoDB what should be the threshold time for slow queries. The `was` field tells us what the earlier profiling level was. Now, let's see a profile log. Execute the following command:

```
mongo> db.system.profile.find()
{ "ts" : ISODate("2012-06-08T07:26:43.186Z"),
  "op" : "query", "ns" : "sodibee_development.authors",
  "query" : { "name" : /in/ }, "nscanned" : 609, "nreturned" : 101,
  "responseLength" : 6613, "millis" : 10, "client" : "127.0.0.1", "user"
  : "" }
```

In the preceding command `op` and `ns` parameters specify the operation and the collection that was profiled. The `query` parameter logs the query that was fired. The `nscanned` parameter specifies the number of objects that were scanned for fetching the result. The `nreturned` parameter specifies the number of objects in the result.



Optimization and performance tuning – tip 1

If you see that the `nscanned` parameter is much higher than `nreturned`, it means that there are a lot of unnecessary objects being scanned.

To resolve this, add an index on these fields used in the search criteria.

Have a look at the previous command a third time:

```
mongo> db.system.profile.find()
{ "ts" : ISODate("2012-06-08T07:26:43.186Z"),
  "op" : "query", "ns" : "sodibee_development.authors",
  "query" : { "name" : /in/ }, "nscanned" : 609, "nreturned" : 101,
  "responseLength" : 6613, "millis" : 10,
  "client" : "127.0.0.1", "user" : ""
}
```

The `responseLength` or `reslen` parameter specifies the number of bytes in the result and the `millis` parameter indicates the time in milliseconds taken by MongoDB for processing this query.

Optimization and performance tuning – tip 2



If you see that `reslen` is huge—a few hundred kilobytes or more—the resultant data being returned is huge and this impacts on the performance. Use the field selector in the `find` method to retrieve only the fields you need.

If I need only the names of authors, we can optimize the query to `db.authors.find({ name: /in/ }, {name: 1})`, so that it will fetch the authors that have an `in` in their name but return only their names and not all the fields. This will reduce the length of the result set.

Using the `explain` function

It's all very well to use the profiler, but that is a reactive measure. That means, we have to analyze existing queries and then optimize them. Is there a way I can take some preventive measures and write an optimized query directly? MongoDB provides the `explain` function to get more information about the performance of the query.

Time for action – explaining a query

Let's say, we want to see how the performance will be for the authors with names that have the `in` search criterion in them. Execute the following query:

```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 20004,
  "nscannedObjects" : 20004,
  "n" : 3037,
  "millis" : 30,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {

  }
}
```

We can see that the previous query was fired in 30 milliseconds. Now let's index the name field and then see the result again. We can index the name field as:

```
>db.authors.ensureIndex({name: 1})
>
```

Now, let's fire the query to find the authors with names that have the `in` search criterion in them again, this time after name has been indexed. Execute the following:

```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BtreeCursor name_1 multi",
  "nscanned" : 20004,
  "nscannedObjects" : 3037,
  "n" : 3037,
  "millis" : 50,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "name" : [
      [
        [
          ""
        ]
      ],
      [
        "/in/",
        "/in/"
      ]
    ]
  }
}
>
```

What just happened?

When we invoke the `explain` function, the query is run and the performance data is calculated. Let's take a deeper look at the query again:

```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 20004,
```

```
"nscannedObjects" : 20004,
  "n" : 3037,
  "millis" : 30,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {

  }
}
```

In this query, MongoDB used the `BasicCursor`, as the name was not indexed then. `nscanned` denotes the number of items, that is, objects and indexes to be examined. `nscannedObjects` denotes the objects examined and `n` is the result. We can see that it takes 30 milliseconds.

Now, if we see that the result after `name` is indexed, we see a different output as follows:

```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BtreeCursor name_1 multi",
  "nscanned" : 20004,
  "nscannedObjects" : 3037,
  "n" : 3037,
  "millis" : 50,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "name" : [
      [
        [
          ""
        ]
      ],
      [
        [
          "/in/",
          "/in/"
        ]
      ]
    ]
  }
}
```

Here we can see that the `BtreeCursor` has been used. We also see a huge difference in `nscanned` and `nscannedObjects`. This is the result of indexing and performance tuning.

Did you notice, however, that the time taken for the indexed query is longer than a basic query! So, did we really optimize the performance?

Yes! Firstly, we ensured that using the index, we have got a far lesser subset of objects. As the number of objects increase, the indexing will become more and more efficient. As we shall soon see in the next section, indexing also reduces querying time!

Using covered indexes

Covered indexes means that all the fields that are being queried and fetched are indexed. If such is the case, the performance of indexed queries becomes excellent! This is because we need not search the documents, only the indexes. As indexes are smaller in size, they can reside entirely in memory and therefore, are accessed very fast.

Time for action – using covered indexes

To test the real power of indexed searches, let's load the database and query during a heavy load. We can easily load the authors using our `fake_authors rake` task as follows:

```
$ rake fake_authors
```

As we know, this will start creating 10,000 more authors. During this time, we shall fire the indexed query and then the covered index query! First we run the indexed query as follows:

```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BtreeCursor name_1 multi",
  "nscanned" : 21695,
  "nscannedObjects" : 3285,
  "n" : 3285,
  "millis" : 248,
  "nYields" : 24,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "name" : [
      [
        "",
        {
          "
```

```
        }
    ],
    [
        /in/,
        /in/
    ]
}
}
```

Now, let's fire the covered indexed query as follows:

```
> db.authors.find({name: /in/}, {_id:0, name:1}).explain()
{
    "cursor" : "BtreeCursor name_1 multi",
    "nscanned" : 27420,
    "nscannedObjects" : 4228,
    "n" : 4228,
    "millis" : 81,
    "nYields" : 19,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : true,
    "indexBounds" : {
        "name" : [
            [
                ""
            ],
            [
                /in/,
                /in/
            ]
        ]
    }
}
```

Notice that the indexed query scanned 21695 objects and took 248 ms and the covered indexed query scanned 27420 but took only 81 ms!

What just happened?

Let's analyze the output results a little more. Have a look at them again:

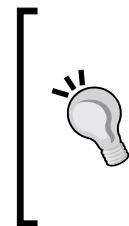
```
> db.authors.find({name: /in/}).explain()
{
  "cursor" : "BtreeCursor name_1 multi",
  "nscanned" : 21695,
  "nscannedObjects" : 3285,
  "n" : 3285,
  "millis" : 248,
  "nYields" : 24,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "name" : [
      [
        [
          ""
        ],
        {
          [
            [
              "/in/",
              "/in/"
            ]
          ]
        }
      ]
    ]
  }
}
```

The `nYields` parameter means the number of times the database lock was yielded—that means it had to yield the lock for a write operation (remember we are creating 10,000 authors). The query completed in 248 ms because of the yields. Now let's see the query for covered indexes as follows:

```
> db.authors.find({name: /in/}, {_id:0, name:1}).explain()
{
  "cursor" : "BtreeCursor name_1 multi",
  "nscanned" : 27420,
  "nscannedObjects" : 4228,
  "n" : 4228,
  "millis" : 81,
  "nYields" : 19,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
```

```
"indexOnly" : true,
"indexBounds" : {
  "name" : [
    [
      [
        "",
        {
          [
            [
              /in/,
              /in/
            ]
          ]
        }
      ]
    ]
  }
}
```

Here, the performance of the query is excellent! What happened here is that MongoDB did not search in the documents but only in the indexes (as `indexOnly` is `true`). It was able to do this because all query fields, as well as the fields to be fetched were indexed! Notice that 27420 objects were scanned in 81 ms and this is a huge performance increase over the earlier query.



Optimization and performance tuning – tip 3

For collections which are fetched very often, index the fields that would be queried and use the `explain` method to check if the query would indeed be fast.

Notice that when using covered indexes, it's imperative to exclude the `_id` field and fetch only the fields that were indexed.

Other MongoDB performance tuning techniques

Now we shall see some more techniques where we can keep checking the performance of operations in MongoDB.

Optimization and performance tuning – tip 4

- ◆ Use the `currentTOP` method to find out the current queries that are in progress.
- ◆ In a shared environment or when using replica sets, enable reads on slaves!

Using mongostat

`mongostat` is a utility that can print the database statistics on the console every second.

The following is what it looks like:

```
$ mongostat -n20
connected to: 127.0.0.1
insert query update delete getmore command flushes mapped vsize res
locked % idx miss % qr|qw ar|aw netIn netOut conn time
      0      0      0      0      0      1      0   208m 3.01g 31m
0      0      0|0      0|0    62b     1k      1 15:04:27
```

As we can see from the output, this prints `insert`, `update`, `delete`, and other basic queries along with a lot more detail!

Understanding web application performance

Achieving high performance from a web application is critical. This is because there are a lot of criteria that determine performance. The following are some of the standard parameters typically considered:

- ◆ Web server response time
- ◆ Throughput
- ◆ User satisfaction – Apdex score
- ◆ Concurrency – **Requests Per Minute (RPM)**
- ◆ Network latency and end-user response

These are only a few parameters that are used for determining web application performance.



Usually if the web server response is under 500 ms and the end-user response is under three seconds, your application is considered to be in good shape.



Web server response time

Web server response is the time taken for any server to respond to an HTTP request. Typically, if we look at the log files that are generated for a Rails application, it gives us some idea about this. The log files would contain something like the following:

```
Started GET "/books" for 127.0.0.1 at 2012-1-28 23:11:35 +0530
...
Completed 200 OK in 359ms (Views: 184.8ms)
```

In the previous code, we can see that a GET request was started and completed in 359ms. Out of this, 184.8ms were spent in rendering HTML. If we are seeing the MongoDB output, we can see other performance metrics—time taken in the database:

```
Sat Jan 28 23:11:35 [conn86] command sodibee_development.$cmd command:  
{ count: "books", query: {}, fields: null }  
  
ntoreturn:1 reslen:48 178ms
```

The web server response obviously includes the time that is spent in the database access too. This is the total time taken by the web server to respond to an HTTP GET request. This does not imply that the user sees the web browser page update so quickly. This means that the web server can respond to this request in about 359ms.

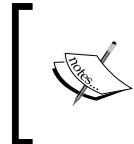
As the data increases, it's quite likely that the response time would increase a bit.

Throughput

The number of simultaneous requests that a web server can handle are called **concurrent requests**. Now, this translates to various factors. Is the web server multithreaded? Does it use a connection pool? Is the web server using evented I/O?

Most web servers are multithreaded. This means that a thread processes every HTTP request that comes to the web server. There is always a limit to the maximum number of threads spawned. Sometimes, web servers use a thread pool and a database connection pool. Basically, these are spawned threads, which process one request at a time. When the request is processed, they don't "die", they simply pick up the next request or wait for one.

New web servers use the reactor pattern to process incoming HTTP requests.



Reactor pattern is a design pattern wherein the system "reacts" to actions. In the case of web servers, a thread is spawned or used for each HTTP request received. In other words, the web server "reacts" by spawning a thread per request.



In any setup, it's pretty difficult to find out the true concurrency of a system. This is typically done in two ways as explained in the following sections:

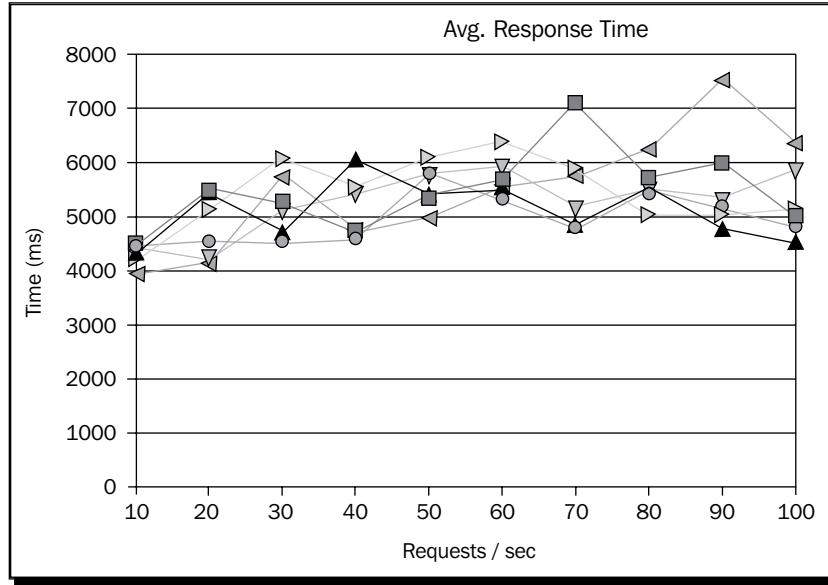
Load the server using httpperf

Bombard the web server with different types of requests using tools such as, httpperf or ApacheBench (ab).

```
httpperf --timeout=10 --client=0/1 --server=<server-name> --port=80  
--uri=/some/uri --wsess=50,5,2 -rate
```

This creates 50 sessions every second, which sends five requests each, after an interval of two seconds. There are plenty of options that can be used with `httperf` that can give various load options.

We can map different response times to a number of requests (shown in different colors in the following graph). `httperf` generates a graph that looks something like the following:

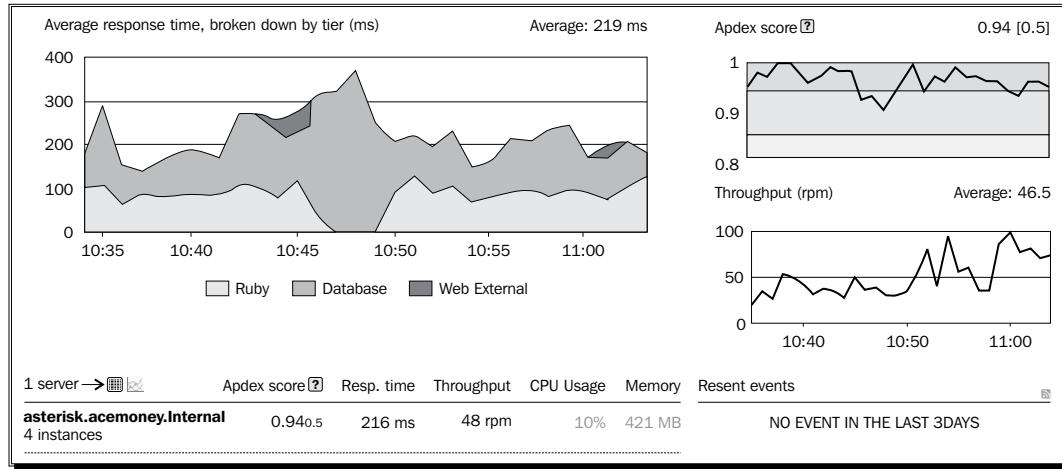


A graph like this tells us the server performance under different loads. From the previously shown data, we can deduce that the average response time is around five seconds and it increases as the load gradually increases from 10 concurrent requests per second to 100 requests per second.

Monitoring server performance

Loading the server seems fine when we have the resources and our web application is already built. However, what can we do if we are building the application? One of the ways of doing this is to continuously monitor the server. There are plenty of ways to monitor server performance but by far the most reliable I have found is RPM from New Relic.

The following is what the dashboard looks like:

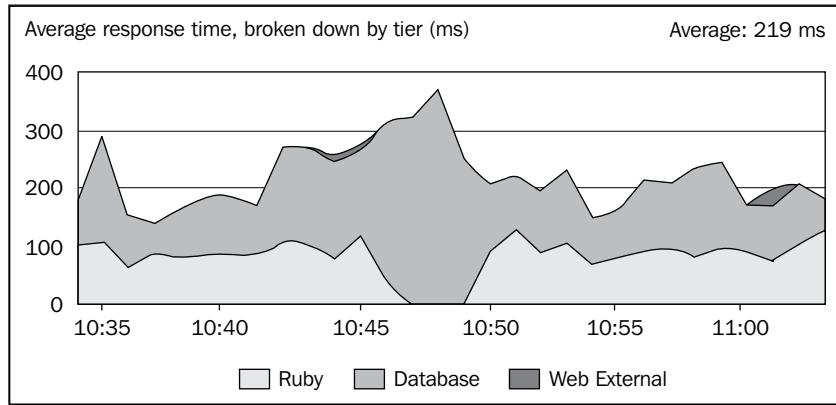


There is a lot of in-depth analysis that it can provide too!

Let's see these in more detail.

Average response time

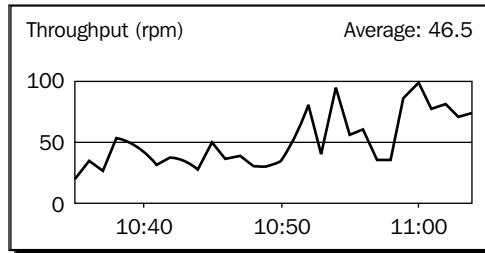
This gives us real time performance metrics as follows:



We can see that the average response time is **219 ms**—with the detailed split of time spent in the database, Ruby processing, and even external calls.

Concurrency/throughput

The throughput is considered in RPM. Considering that requests per second would virtually be the profiling request itself, it would kill the throughput results. So it's easier to average the results over a minute:

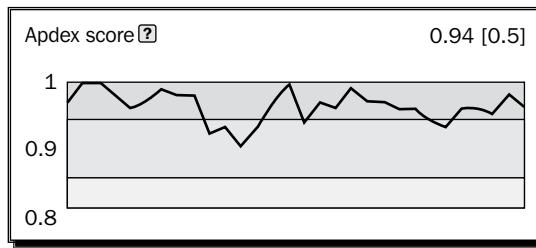


This tells that the average RPM is **46.5**. This tells us the real-time concurrency of the system.

Apdex Score

Apdex is the short name for **Application Performance Index**. There are various ways and different means to identify the Apdex. New Relic defines the Apdex on a percentage scale. So, the closer the Apdex is to 1, the better the application performance.

Apdex scores are samples taken from real time requests per minute and distributed into different categories such as Satisfied, Tolerating, and Frustrated:

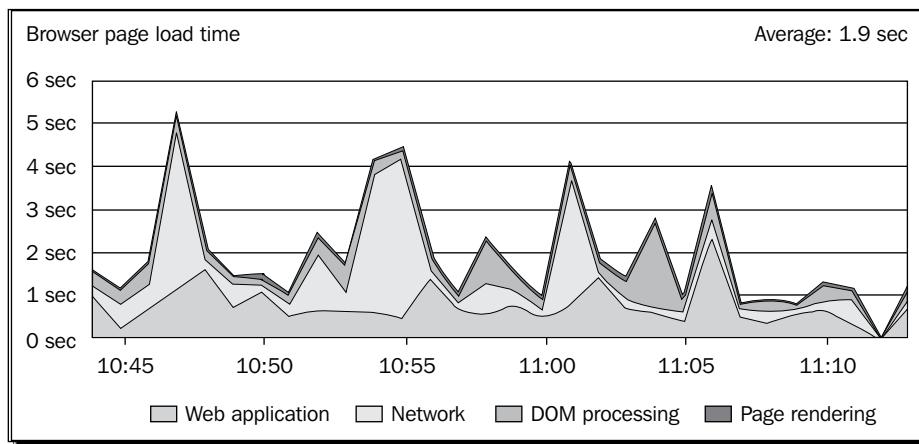


Finally, we can always see a summary of what's happening in real time, shown as follows:

Apdex score	Resp. time	Throughput	CPU Usage	Memory
0.940.5	216 ms	48 rpm	10%	421 MB

End-user response and latency

A server response time is not always enough. We also want to ensure that our end-user web page has refreshed in the proper time. Typically, end-user response under three seconds is considered decent:



The preceding screenshot shows us that the average page rendering time was 1.9 seconds. If we also look closely, the maximum colored area is the network latency!

Optimizing our code for performance

Now that we have seen what performance is all about, let's see how we can tune our application with MongoDB for better performance.

Indexing fields

As we have seen earlier, using indexes increases the performance quite a lot, especially for reads! Indexes are stored in binary trees. Remember that indexes require more storage computation due to an addition of information to B-trees during inserts and removal of data from B-trees during deletes. This makes the inserts and updates fractionally slower.

However, in a typical web application there is always a lot more data retrieval than updates, so using indexes judiciously makes sense.

[ Do not use indexes for write-intensive operations, as they would be counter productive!]

Optimizing data selection

Even though indexes help increase performance, it never harms in taking a few good practices to ensure the database is not over loaded and hence available for more requests, which in turn increases overall performance.



Never fetch all the documents in a collection. Use pagination and limit to a convenient number depending on your application.



Remember, that as a web application usually has a long life, data would grow! So, if you keep fetching all the fields in a document all the time, we would be degrading the performance over time.



Fetch only the fields you require if we are not caching anything.



If you don't require the entire document, why fetch all of it? However, if you couple this with a caching strategy, it makes sense to actually fetch the entire document. As we shall see later about caching strategies, it pays to fetch the entire document when working in a Rack application with caching enabled.

Optimizing and tuning the web application stack

We have seen how to tune a database and what web application performance is all about. There's more! We can tune our Ruby web application to enhance the performance further. Ruby, when used in conjunction with the right application stack can make a world of difference.

Performance of the memory-mapped storage engine

This is the default storage engine used by MongoDB and is enabled by default. It uses memory-mapped files for its disk I/O. This gives advantages of memory-like speeds and also ensures that the file system cache and the database cache are the same!

As MongoDB uses the standard memory-mapped files, the operating system's virtual memory manager takes care of the size, swapping, and management of these files. As the OS virtual memory manager is updated, it automatically boosts MongoDB's performance. That means, two benefits for the price of one!

Choosing the Ruby application server

A web server is one that processes HTTP requests. Some of the popular web servers are Apache and nginx. However, the request could be processed by different application servers—PHP, Java, Ruby, or similar ones. Once the request is sent to the application server, it needs to process it quickly. The performance of these application servers is critical.

There are plenty of Rails application servers available. All these application servers are Rack applications, so it's very convenient to switch between them. At the time of writing this book, these are the currently available and recommended choices for web servers.

Passenger

This is a library that compiles nicely with Apache or nginx. A Rack application can be easily configured to run a Sinatra or Rails application. The library needs to be complied and loaded at runtime. Passenger spawns and reaps worker processes depending on the load on the web server. This makes it a very powerful choice for scalable web servers.

Mongrel and Thin

Mongrel is a web server that processes Rails requests. Thin is Mongrel plus evented I/O and Rack bundled together. The number of worker processes can easily be configured. Both are very fast and very efficient. We can configure various options with this, including the maximum number of connections per worker.

Unicorn

Unicorn is known for its stability and reliability. It is relatively newer than the others but addresses issues such as respawning on failures and preempting slow requests. It uses the Unix domain sockets for load balancing instead of HAProxy in the case of Thin or Mongrel.

All these web servers are really good for deploying Ruby web applications and they significantly improve the performance of the application.

Increasing performance of Mongoid using bson_ext gem

`bson_ext` gem is a C extension to accelerate BSON serialization. This significantly increases the performance. It is used in conjunction with `mongoid` and `bson` gems and is highly recommended.

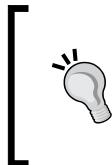
Caching objects

When we fetch information from the database, we can store it in the memory for some time—called the **time to live(TTL)**. So, in case we need to fetch the same object again, instead of querying the database, we look up the cache. This increases performance, as a memory read is much faster than a database read (which is disk I/O). This also keeps a lesser load on the database.

When we have a caching layer enabled, this is how data is fetched:

- ◆ Look up the cache for the object
- ◆ If found, return it
- ◆ If not found, look up the database and fetch the data
- ◆ Save it to cache and return it

Some caching strategies even allow "lazy writes". This means that we can use caching not just for reads but also for updates! When an object is updated, we update it in memory, mark it to be updated, and return the response immediately. This has a tremendous performance boost and this information is written to the database later, typically a few seconds later. So, if we have a thousand increments to an object, not only is it faster and gives better performance, the lazy write ensures that writes to the database are optimized and aren't done for each change of the object.



Remember that this "eventual consistency" would not be the right choice for very heavy transaction-related web applications. So, we should choose a caching strategy carefully.

It's also very important to remember that we fetch the entire document from the database when we cache them as objects.

Memcache

Instead of using the system memory for the caching, we can alternatively set up a memcache server and configure the Rack application to use this for caching! This is the recommended and standard practice for large scale web-based applications.

Redis server

Redis is an in-memory database that can be used as an object cache. As it guarantees atomic updates and lazy persistence, it is also an excellent choice. Remember that it adds one more point of failure in the stack, so it should be monitored. Moreover, Redis also consumes memory, so remember to have a good memory bank (of at least 1 GB or 2 GB) in large-scale production systems.

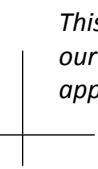
Summary

In this chapter we have learned the concept of web application performance and seen the different parameters considered when we evaluate a web application. We tuned MongoDB queries for performance using indexes and covered indexes. We saw how we can tune the database and what MongoDB already provides to ensure that performance is good. We also saw how we can optimize our Ruby web application by making the right choice of web servers and an appropriate object caching strategy.

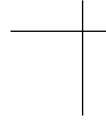
In the next chapter, we shall build the entire web application making use of Ruby, Rack, and MongoDB via Mongoid. This would be pretty exciting as we shall finally see things taking shape and it should be satisfying!

8

Rack, Sinatra, Rails, and MongoDB – Making Use of them All



This is a web development guide! Until now, we have been reinforcing our concepts! Building the data models and control logic is the core of the application. Now we shall put all these pieces together in a web application.



In this chapter we will learn the following:

- ◆ Modeling objects in Sinatra and Rails
- ◆ Building the logic and control flow
- ◆ Designing the Views – web interface
- ◆ Testing web applications
- ◆ Documenting our code

This chapter will explain in detail how a Rack application is built. We shall touch upon some interesting tools, such as RSpec for testing and YARD for documentation. But we shall only skim these concepts, as these are concepts for which there are books available.

By the end of this chapter, we shall have a full-fledged web application up and running in Sinatra and in Rails.

Revisiting Sodibee

We have played around with some aspects of Sodibee, such as Book, Author, and Category. Now, we shall build the full-fledged web application in Rails and Sinatra. This is what we are going to do – it's what we started out with, and a little more—The Sodibee (pronounced as |saw-d-bee|) Library Manager.

Books belong to categories like Fiction, Non-fiction, Romance, Self-learning and so on. Books have one author and one publisher. Books can be rated and reviewed.

Books can be leased or bought. When books are bought or leased, the customer's details (such as name, address, phone, and e-mail) are registered, along with the list of books purchased or leased. A ledger is maintained on the quantity of each book sold and the number of times it was leased.

The Rails way

Rails is an amazing framework when it comes to evolution! It evolves at a rapid pace and there are so many new components available to plug into Rails, that we could be left overwhelmed! For our application, we shall use the following components:

- ◆ Rails 3.2.2 (the latest version currently available)
- ◆ Ruby 1.9.3
- ◆ MongoDB using the `mongoid` gem
- ◆ The Twitter Bootstrap framework for the UI
- ◆ Haml for Views
- ◆ Sass for all our CSS work
- ◆ CoffeeScript for all our JavaScript work
- ◆ jQuery (the default JavaScripting option)
- ◆ `simple_form` and `nested_form` for HTML forms

Wow! Has this become a little exhaustive? Don't worry, as we will shortly see, Rails is all about "convention over configuration" and by using the right tools for the right job, you end up writing very little code for a lot of functionality!

Setting up the project

We have already seen this a couple of times. Here it is in brief again:

```
$ rails new sodibee -JO
```

Following is the Gemfile that we shall use:

```
source 'https://rubygems.org'

gem 'rails', '3.2.2'      # Rails Version.

gem 'mongoid'              # MongoDB config
gem 'bson'
gem 'bson_ext'

gem 'haml'                  # Templating markup
gem 'haml-rails'

gem "jquery-rails"          # jQUREy config

# Need nested form from the git repos to ensure it's the latest one
gem "nested_form", :git => 'git://github.com/ryanb/nested_form.git'
gem 'simple_form'

# Rails Asset pipeline
group :assets do
  gem 'sass-rails',    '~> 3.2.3'    # Sass
  gem 'coffee-rails',  '~> 3.2.1'    # CoffeeScript
  gem 'bootstrap-sass', '~> 2.0.1'    # Bootstrap
  gem 'uglifier',     '>= 1.0.3'
end

group :development, :test do
  gem 'rspec-rails'
  gem 'spork'           # speedy testing!
end
```

As you can see, we have gems for MongoDB, Haml, Sass, Bootstrap and even jQuery. `nested_form` and `simple_form` (as we shall see later) are very useful gems for HTML forms.

Let's update the bundle for this Rails project:

```
$ bundle install
$ rails g mongoid:config
```

Remember to remove `activerecord` from the `config/application.rb` file. This is how the `config/application.rb` file should look like:

```
require "action_controller/railtie"
require "action_mailer/railtie"
require "active_resource/railtie"
require "sprockets/railtie"
```

Modeling Sodibee

While we look at these models, we shall also learn a few Rails concepts along the way!

Time for action – modeling the Author class

First let's write the `Author` model. We do it as follows:

```
class Author
  include Mongoid::Document

  field :name, type: String

  validates_presence_of :name

  has_one :address, as: :location, autosave: true, dependent: :destroy
  has_many :books, autosave: true, dependent: :destroy

  accepts_nested_attributes_for :books, :address, allow_destroy: true
end
```

What just happened?

An author has many books and has one address. This is declared as follows:

```
class Author
...
  has_many :books, autosave: true, dependent: :destroy
  has_one :address, as: :location, autosave: true, dependent: :destroy
...
end
```

We have already seen relationships via Mongoid, but here are a few more options:



- ◆ `:autosave`: This option is specified in the parent model and enables its associated child objects to be saved along with the parent
- ◆ `:as`: This is the polymorphic relation
- ◆ `:dependent`: This option is also specified on the parent model and ensures that the dependent child objects are destroyed when the parent is destroyed

When we are creating an author, we would also like to update all the books written by the author as well as update his address. We do this by accepting nested attributes:

```
class Author
...
  accepts_nested_attributes_for :books, :address, allow_destroy: true
...
end
```

As the name suggests, `accepts_nested_attributes_for` accepts nested attributes for the child relation.



We can only accept nested attributes for children. That means we should use them only in the parent relation.

We shall see how this comes into play when we build the Views.

Update the `Author` model as follows:

```
class Author
  ...
  validates_presence_of :name
  ...
end
```

Because this is a Mongoid document, it has all the features that are available with `ActiveModel`, such as `ActiveModel::Validations`. So we can use all the available validations here. In this case, we validate the presence of the `name` to ensure that an `Author` object is not created without the name!

Time for action – writing the Book, Category and Address models

Now let's take a look at the remaining models. The `Book` model is as follows:

```
# app/models/book.rb

class Book
  include Mongoid::Document

  field :title, type: String
  field :publisher, type: String
  field :published_on, type: Date
  field :price, localize: true
  field :votes, type: Array

  validates :title, presence: true

  belongs_to :author
  has_and_belongs_to_many :categories

  embeds_many :reviews
end
```

Now let's add the Category and Address model:

```
# app/models/category.rb

class Category
  include Mongoid::Document

  field :name, type: String

  has_and_belongs_to_many :books
end

# app/models/address.rb

class Address
  include Mongoid::Document

  field :street, type: String
  field :zip, type: Integer
  field :city, type: String
  field :state, type: String
  field :country, type: String

  belongs_to :location, polymorphic: true
end
```

What just happened?

Nothing that we didn't already know! We have seen all these fields and relations in the earlier chapters! Remember that Address has a polymorphic relation as it can be related to any other model!

Time for action – modeling the Order class

Now, let's look at a few new aspects! An order is of two types; either a lease or a purchase. The Order model can be written as follows:

```
# app/models/order.rb
class Order
  include Mongoid::Document

  field :created_at, type: DateTime
  field :type, type: String # Lease, Purchase

  belongs_to :book
  belongs_to :member
```

```
embeds_one :lease
embeds_one :purchase
end
```

The Purchase model can be written as follows:

```
# app/models/purchase.rb
class Purchase
  include Mongoid::Document

  field :quantity, type: Integer
  field :price, type: Float

  embedded_in :order
end
```

The Lease model can be written as follows:

```
# app/models/lease.rb
class Lease
  include Mongoid::Document

  field :from, type: DateTime
  field :till, type: DateTime

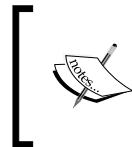
  embedded_in :order
end
```

What just happened?

Here we are following the standard paradigm for a `type` field. If the `type` is `:lease`, we shall look up the `Lease` embedded object. If it's `:purchase`, we shall look up the `Purchase` embedded object. We could have made this polymorphic, but then how will we learn the different ways of coding?

Understanding Rails routes

What are routes, did you say? They are the URLs that we shall use to access the application from the web browser. Rails goes one step further and sets up RESTful routes by default.



REST stands for **R**Epresentational **S**tate **T**ransfer. It represents resources and actions performed on them. Given a combination of the resource, HTTP verbs (GET, PUT, POST and DELETE) and some basic actions, we can define standard operations.

What is the RESTful interface?

RESTful interfaces are the definition of resources from which the URLs are generated. We can understand this better from the following table:

HTTP Verb	Author Resource URL	Controller Action	Description
GET	/authors	:index	List all Authors
GET	/authors/:id	:show	Show Author details
GET	/authors/:id/edit	:edit	Show the edit Author form
PUT	/authors/:id	:update	Update Author
POST	/authors	:create	Create Author
GET	/authors/new	:new	Show the new author form
DELETE	/authors/:id	:destroy	Delete an Author

Time for action – configuring routes

We can invoke different URLs depending on the action we want to perform. We configure the routes for our application in `config/routes.rb`:

```
Sodibee::Application.routes.draw do  
  
  resources :authors do  
    resources :books  
  end  
  
  resources :orders  
  resource :categories  
  
  root :to => 'authors#index'  
end
```

What just happened?

These are the basic routes. Let's see them one by one:

```
Sodibee::Application.routes.draw do  
  
  resources :authors do  
    resources :books  
  end  
  
  resources :orders  
  resource :categories  
  
  root :to => 'authors#index'  
end
```

The highlighted line of code in `config/routes.rb` generates various routes. We can see them by issuing the following command:

```
$ rake routes

categories      POST   /categories(.:format)    categories#create
new_categories  GET    /categories/new(.:format)   categories#new
edit_categories GET    /categories/edit(.:format)  categories#edit
                  GET    /categories(.:format)    categories#show
                  PUT    /categories(.:format)    categories#update
                  DELETE /categories(.:format)  categories#destroy
```

As we can see, different HTTP verbs and the URLs map to different actions. Here `categories` is a resource. Just like we have resources, we also have nested resources; for example, books cannot exist without an author. Have a look at the following:

```
Sodibee::Application.routes.draw do

  resources :authors do
    resources :books
  end

  resources :orders
  resource :categories

  root :to => 'authors#index'
end
```

Here, books can be accessed only in the namespace of the author. So, this builds URLs like this: `/authors/:author_id/books/:id`.

Understanding the Rails architecture

This is a good time to explain how a Rails request is processed. As you are probably aware, Rails follows the **Model-View-Controller (MVC)** architecture, that is, it follows the MVC design pattern. The aim of this architecture is to divide the application into more than just one long procedural program!

The Model holds all the data manipulation code. Typically, most of the code resides in the models. The data validations, relationships, pre and post processing of data, pre and post action callbacks are written in models. Models should be fat!



Domain-Driven Design by Eric Evans is an excellent book that talks about writing code, based on domain logic and organizing the complexity. In Rails terminology, we extensively use modules and include them in the models to keep models thin and keep the domain logic separate.

The Controller controls the flow for processing the request. Authentication and authorization checks are done here. The flow of control on an action's success or failure is written here. For example, what should be done if an object cannot be saved or updated? It also has pre and post action filters. Controllers should be skinny!

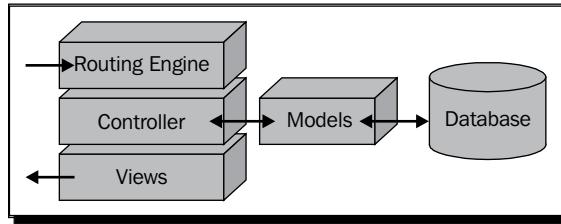
The View is the final HTML that is rendered. Writing raw HTML can be very tedious, so it's usually managed via templates—ERB, Haml, Liquid, Jade, Slim, and so on. These are the template markup languages that generate HTML and can also process Ruby embedded in them. Haml is what we shall be using. Views should avoid processing code as it impacts the performance drastically. They should typically only access data, as Ruby instance variables or JSON.

The Helper is a module that helps the Views process Ruby code in a cleaner way. Suppose we need to manipulate some data, rather than writing it in the View, it should be written in the Helper. This also avoids rewriting code and obeys the **Don't Repeat Yourself(DRY)** principle.

I'll say it again "Don't Repeat Yourself", "Don't Repeat Yourself"! (Just couldn't resist repeating myself here!)

Processing a Rails request

Ever wondered what really happens when a Rails request is received? With so many different components floating around, how are these pieces of the puzzle put together? The following diagram should clear things for you:



A Rails request is processed as follows:

- ◆ When a Rails request comes to the web server, the Rack (remember?) identifies the HTTP Verb, the request parameters, and the URI (the string after the host name). For example, if we type the URL `http://localhost:3000/authors/new` in the browser's address bar, the Rails server will identify this as a GET request with the URI as /authors and as there are no parameters passed, the `params` will be an empty hash.
- ◆ Now, the Rails web server resolves the URI and maps it to a Controller and an action. It parses the URI and maps it to a URI format as seen in the `rake routes` command. As we can see, this will map to the `Authors#index` action. We shall see more detailed examples shortly.

- ◆ Now, we know the Controller name (`AuthorsController`) and the action (`index`). An `AuthorsController` object is created for this request and the `index` action is invoked on that object. With that, we are now in the Controller code!
- ◆ The Controller's action now processes the request and accesses the Models and gathers the information required.
- ◆ Now, when it's time to send back a response, just as the Controller and action were resolved, we need to find the template for this action. It would reside in the `views/<controller name>/<action template>` and in our example, it would be `views/authors/index.html.haml`.
- ◆ Here lies the "Rails magic" (very rarely explained in Rails books). After the Controller processing is done, the Rails web server creates an instance of the `ActionView` object (it's a class which helps in rendering) and copies all the instance variables from the Controller object we created into this object. Yes! That's right, we can copy instance variables from one object to another.
- ◆ Now, we pass the template file to this object and process it along with direct access to the instance variables! Voila – the output is an HTML response.

Tips to ensure higher efficiency and productivity in your code

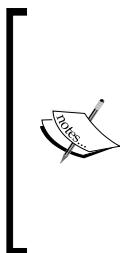


- ◆ Try never to fetch too much data in your Controller's instance variables. If there are 100,000 objects fetched from the database, not only is it heavy on memory but also it would mean we have to copy these 100,000 objects into the View, which can be expensive. Use pagination!
- ◆ Don't keep unnecessary instance variables in the Controller. Create only those instance variables that will be accessed in the Views.
- ◆ Ensure that models are not accessed from the Views. Understandably, this will reduce efficiency because data access from the Views means database I/O!

Coding the Controllers and the Views

Here is where our web application kicks in. Let's write some Controllers first. Every Rails application has the default Controller as `ApplicationController`. For example, consider the following:

```
class ApplicationController < ActionController::Base
  protect_from_forgery
end
```



`protect_from_forgery` is a method which uses the Cross Site Request Forgery (CSRF) token to ensure that the data is being posted from a secure form.

There are more ways to secure a Rails application. Recently, a mass assignment vulnerability was found and resolved using `attr_accessible` but not before the mighty Github portal was hacked. (<http://github.com/blog/1068-public-key-security-vulnerability-and-mitigation>)

Time for action – writing the AuthorsController

Now we shall see what the Authors Controller has in store for us. Have a look at RESTful routes again and remember that all the RESTful actions are methods in the Controller class. Have a look at the `AuthorsController`:

```
# app/controllers/authors_controller.rb

class AuthorsController < ApplicationController

  # GET /authors
  def index
    @authors = Author.all.includes(:books)
  end

  # GET /authors/new
  def new
    @author = Author.new
    @author.build_address
    @author.books.build
  end

  # POST /authors
  def create
    @author = Author.new(params[:author])
    @author.save!
    redirect_to authors_path, notice: "Author created successfully"

    rescue
      render :new
  end

  # GET /authors/:id/edit
  def edit
```

```
    @author = Author.find(params[:id])
    @author.build_address unless @author.address
    @author.books.build if @author.books.empty?
end

# PUT /authors/:id
def update
    @author = Author.find(params[:id])
    if @author.update_attributes(params[:author])
        redirect_to authors_path, notice: "Author updated successfully"
    else
        render :edit
    end
end
end
```

It's still too early to run and test this code. We need to build the Views before we can see something in the browser!

What just happened?

Let's take a look at the `index` method:

```
# GET /authors
def index
    @authors = Author.all.includes(:books)
end
```

The preceding method lists all the authors. (We are ignoring pagination here and fetching all the authors.) As we need to render the author objects in the Views, we are storing them in an instance variable `@authors`.

Solving the N+1 query problem using the includes method

`includes` is a method that does "eager loading" of associated objects. Suppose we want to show the author names and the book titles for that author, we would need to fetch the `Book` object for each author.

The inefficient way to do this is to only fetch the `Author` object and then on-demand, fetch the `Book` object when needed. This means that if there are 100 authors, we will be firing 101 queries – one for fetching all the authors and one query for fetching books for each author! This is indeed expensive. This is also popularly called the N+1 query problem!

The efficient way of doing this is by firing one query to fetch the authors and only one more query to fetch all the books of the selected authors. So, whether I have 10 authors or 100,000 authors, I will always fire only two queries!

Alright! Let's get back to the code now. Now let's see the `new` and `create` methods:

```
# GET /authors/new
def new
  @author = Author.new
  @author.build_address
  @author.books.build
end

# POST /authors
def create
  @author = Author.new(params[:author])
  @author.save!
  redirect_to authors_path, notice: "Author created successfully"

rescue
  render :new
end
```

The `new` and `create` methods are used in tandem. In the `new` method, what's important to see are the following two lines used for building the related objects:

```
# GET /authors/new
def new
  @author = Author.new
  @author.build_address
  @author.books.build
end
```

Hey! We haven't even saved an object to the database, so how are we relating them? That's the beauty of Rails relations. When the `Author` object is created, it does not mean it's saved to the database. When the `save` is called in the `create` method, it is actually persistent in the database!

Relating models without persisting them

Did I hear you ask, what's the difference between `build_address` and `books.build`? Why not `build_books` or `address.build`? Here it goes!

As the `Author` model has only one address (the `has_one` relation), we can call a method directly – `build_address`. If this were `@author.address.build`, it would throw an exception saying **build call on nil object**. As the `Author` model has many books (the `has_many` relation) it's internally stored as an empty array. So we can call `@author.books.build` on it.

Hey! What does `.build` do anyway? How is it different from `new`? Another good question!

When we create an object using `new`, it has an `id` that is not saved to the database (yet). We can use `.build` to create an associated objects in memory using the relations even on these objects that are not in the database.



`@author.books.build` and `@author.books.new` are equivalent, as `books` is an array because of the `has_many` relation!

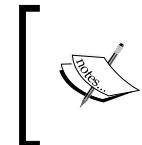


Back to our code again. Let's have a look at the code for the `POST` request:

```
# POST /authors
def create
  @author = Author.new(params[:author])
  @author.save!
  redirect_to authors_path, notice: "Author created successfully"

  rescue
    render :new
end
```

For creating an author, we require a `POST` request to `/authors!` If all the validations pass (such as, name of author is present), the `@author` instance variable is instantiated. When we call the `@author.save!` it is actually saved to the database!



"Bang methods" such as, `save!` or `create!` have a special meaning.
An exception will be raised in case the object cannot be persisted.
`save` and `create` can also be invoked but they do not raise an exception. They simply return `true` or `false`.



If anything goes wrong in the preceding method, an exception will be raised and the `Author` object will have its `errors` field populated! On the basis of this `errors` field, we can show relevant error messages in the browser. We shall soon see in the Views, what the Rails framework does for us "automagically".

If the object is successfully saved to the database, the Controller redirects the request to the author's index page!

Let's see the `edit` and `update` methods now:

```
# GET /authors/:id/edit
def edit
  @author = Author.find(params[:id])
  @author.build_address unless @author.address
```

```
    @author.books.build if @author.books.empty?  
  end  
  
  # PUT /authors/:id  
  def update  
    @author = Author.find(params[:id])  
    if @author.update_attributes(params[:author])  
      redirect_to authors_path, notice: "Author updated successfully"  
    else  
      render :edit  
    end  
  end
```

This is similar to the `new` and `create` methods, except that we search for the relevant object from the database using the `find` method.

Notice the `:id` in the route `/authors/:id/edit`. How did we access it from `params`? Hey! What are these `params`?

 params is a hash stored in the `HTTPRequest` object and accessible to the Controller method that is invoked. `params` contains all the route parameters, (such as `:id`, the one we just saw), the GET parameters, (such as, `?foo=bar` in the URL) and the POST parameters (from the HTTP forms). So we don't have to do any special handling to fetch parameters, they are already there for us. Thank you Rack!

The `update` method also shows us an interesting idiom:

```
# PUT /authors/:id  
def update  
  @author = Author.find(params[:id])  
  if @author.update_attributes(params[:author])  
    redirect_to authors_path, notice: "Author updated successfully"  
  else  
    render :edit  
  end  
end
```

Instead of using `save!` or `update!` we are using the return value of `update_attributes` and testing it for `true` or `false`. If the object is saved successfully to the database, the control should redirect to the Author's index otherwise, it should render the `edit` action with the `@author` object errors to indicate the error messages.

Designing the web application layout

Finally, we shall now learn how to render the data we have collected in a neat and clean way! Welcome Bootstrap and Haml!

Late in 2011, Twitter released a framework called Bootstrap. It's a bunch of CSS and JS files. They are unobtrusive and integrated with jQuery. They even have a responsive design! (that is, it would work on all media—phones, tablets, and the web.)

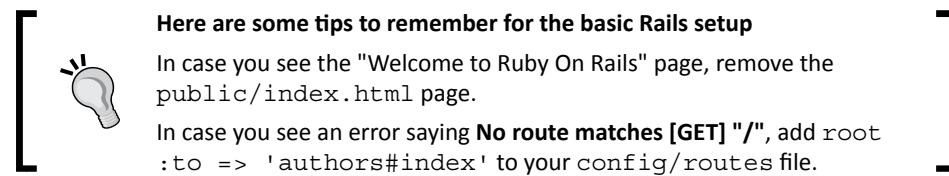
The layout of an application is the base page design. It has a header, content, and footer. Let's design this!

Time for action – designing the layout

Start your engines! Let's start the server:

```
$ rails s
=> Booting WEBrick
=> Rails 3.2.2 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
INFO  WEBrick 1.3.1
INFO  ruby 1.9.2 (2011-07-09) [i386-darwin9.8.0]
INFO  WEBrick::HTTPServer#start: pid=15943 port=3000
```

Now type `http://localhost:3000` in the browser's address bar and we are on our way!



Here is our layout, it's "bootstrapped". This is how our `app/views/layouts/application.html.haml` looks:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"}/
    %meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"}/
      %title Sodibee Library Manager
```

```
= javascript_include_tag "application"
= stylesheet_link_tag "application"
= csrf_meta_tags

%body
  .navbar
    .navbar-inner
      .container-fluid
        = link_to "Sodibee", root_path, :class => 'brand'
      %ul.nav
        %li.dropdown
          %a.dropdown-toggle{ :href => '#', "data-toggle" => "dropdown" }
            ="Authors"
            %b.caret
          %ul.dropdown-menu
            %li= link_to "List Authors", authors_path
            %li= link_to "New Author", new_author_path
            %li= link_to "Orders", orders_path
            %li= link_to "New Order", new_order_path
      .container
        .content
          = yield
      .footer
        %p Packt Publishing &copy; Company 2011
```



In case you see the app/views/layouts/application.html.erb file, you can simply remove it. We are using Haml and not ERB.



In case you see an error Missing template authors/index, simply add an empty file app/views/authors/index.html.haml. We can add the Haml code into it later.

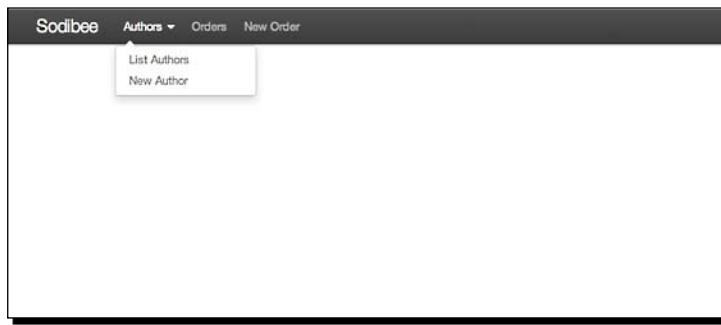
We also have to configure the JavaScript and CSS via the Asset pipeline. Let's take a look at the main JavaScript file app/assets/javascript/application.js:

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require_tree .
```

And now, let's configure our stylesheets. In case there is already an app/assets/application.css file, remove it entirely and add a new file app/assets/application.css.sass with the following contents:

```
@import 'bootstrap'
```

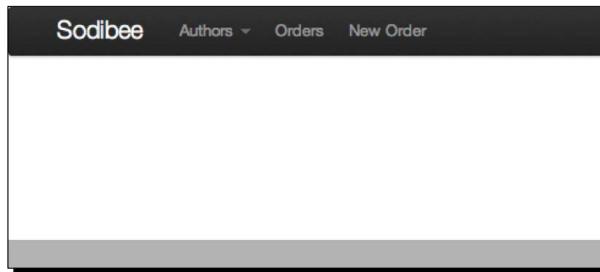
Now, type the URL `http://localhost:3000` in the browser's address bar and you should see our application with a very neat and fancy layout, shown as follows:



What just happened?

Rails Magic! That's what just happened. Let's study this in detail.

A closer look at the Top Navigation bar reveals that **Authors** is a drop-down menu with two more options: **List Authors** and **New Author**. This was all coded in Haml:



Haml is an indentation-aware templating language. It looks neat and tidy and you can find a lot more information at <http://haml-lang.com>.

A very quick Haml reference can be explained as follows:

% adds HTML tags like `span`, `div`, `p` and so on.

. adds the `class` attribute to `div` tag. For example, `.footer` creates the `<div class="footer">` tag.

adds the `id` attribute to the `div` tag. For example, `#authors` creates the `<div id="authors">` tag.

Both can be used in tandem. For example, `#authors.well` creates the `<div id="authors" class="well">` tag.

= suffix implies Ruby code processing. For example, `%p= 1 + 1` creates `<p>2</p>`.

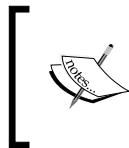


Now let's see the code in application.html.haml:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"/>
    %meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"/>
    %title Sodibee Library Manager
    = javascript_include_tag "application"
    = stylesheet_link_tag "application"
    = csrf_meta_tags

  %body
    .navbar
      .navbar-inner
        .container-fluid
          = link_to "Sodibee", root_path, :class => 'brand'
        %ul.nav
          %li.dropdown
            %a.dropdown-toggle{ :href => '#', "data-toggle" => "dropdown" }
              ="Authors"
              %b.caret
            %ul.dropdown-menu
              %li= link_to "List Authors", authors_path
              %li= link_to "New Author", new_author_path
              %li= link_to "Orders", orders_path
              %li= link_to "New Order", new_order_path
        .container
          .content
            = yield
        .footer
          %p Packt Publishing &copy; Company 2011
```

We just saw the core HTML header generation. We can define HTML meta tags here, as well as the default title of the page and load JavaScript and CSS! The CSRF token is added here by default as a security measure.



The `%meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"}/` gets Bootstrap to configure the Views as a responsive layout, that is these pages will be seen properly aligned on any device—a computer monitor, an iPhone, or any mobile, or touch device.

Take a look at the preceding Haml code again:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"/>
    %meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"}/
      %title Sodibee Library Manager
      = javascript_include_tag "application"
      = stylesheet_link_tag "application"
      = csrf_meta_tags

  %body
    .navbar
      .navbar-inner
        .container-fluid
          = link_to "Sodibee", root_path, :class => 'brand'
          %ul.nav
            %li.dropdown
              %a{:class => 'dropdown-toggle', :href => '#', :data => {:toggle => 'dropdown'}}
                ="Authors"
                %b.caret
                %ul.dropdown-menu
                  %li= link_to "List Authors", authors_path
                  %li= link_to "New Author", new_author_path
                  %li= link_to "Orders", orders_path
                  %li= link_to "New Order", new_order_path
        .container
          .content
            = yield
        .footer
          %p Packt Publishing &copy; Company 2011
```

In the preceding code, the highlighted part is the navigation bar—the black bar that we see! We can define our application logo there, as shown in the following code:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"/>
    %meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"}/
      %title Sodibee Library Manager
```

```
= javascript_include_tag "application"
= stylesheet_link_tag "application"
= csrf_meta_tags

%body
  .navbar
    .navbar-inner
      .container-fluid
        = link_to "Sodibee", root_path, :class => 'brand'
      %ul.nav
        %li.dropdown
          %a{:class => 'dropdown-toggle', :href => '#', :data =>
{ :toggle => 'dropdown' }}
            ="Authors"
            %b.caret
            %ul.dropdown-menu
              %li= link_to "List Authors", authors_path
              %li= link_to "New Author", new_author_path
              %li= link_to "Orders", orders_path
              %li= link_to "New Order", new_order_path
      .container
        .content
          = yield
      .footer
        %p Packt Publishing &copy; Company 2011
```

The highlighted part of the code is a drop-down menu bar, as we can see in our application. Let's now see the Haml code for the **Orders** drop-down menu bar:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"}/
    %meta{:content => "width=device-width, initial-scale=1.0", :name => "viewport"}/
    %title Sodibee Library Manager
    = javascript_include_tag "application"
    = stylesheet_link_tag "application"
    = csrf_meta_tags

  %body
    .navbar
      .navbar-inner
        .container-fluid
          = link_to "Sodibee", root_path, :class => 'brand'
```

```
%ul.nav
  %li.dropdown
    %a{:class => 'dropdown-toggle', :href => '#', :data =>
{:toggle => 'dropdown'}}
      ="Authors"
      %b.caret
      %ul.dropdown-menu
        %li= link_to "List Authors", authors_path
        %li= link_to "New Author", new_author_path
        %li= link_to "Orders", orders_path
        %li= link_to "New Order", new_order_path
.container
  .content
    = yield
  .footer
    %p Packt Publishing &copy; Company 2011
```

And the highlighted statements are standard top-level menu items!

Have a look at the code for the `yield` method:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"/>
    %meta{:content => "width=device-width, initial-scale=1.0", :name =>
"viewport"/>
      %title Sodibee Library Manager
      = javascript_include_tag "application"
      = stylesheet_link_tag "application"
      = csrf_meta_tags

  %body
    .navbar
      .navbar-inner
        .container-fluid
          = link_to "Sodibee", root_path, :class => 'brand'
        %ul.nav
          %li.dropdown
            %a{:class => 'dropdown-toggle', :href => '#', :data =>
{:toggle => 'dropdown'}}
              ="Authors"
              %b.caret
              %ul.dropdown-menu
                %li= link_to "List Authors", authors_path
```

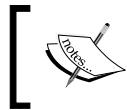
```
%li= link_to "New Author", new_author_path  
%li= link_to "Orders", orders_path  
%li= link_to "New Order", new_order_path  
.container  
.content  
= yield  
.footer  
%p Packt Publishing &copy; Company 2011
```

This is where the dynamic code is rendered! `yield` is a Ruby method that renders any block of code passed. All the code that we want to dynamically change and render in this layout is automatically passed as a block of Haml with Ruby code embedded in it!

Understanding the Rails asset pipeline

Rails 3.1 introduced the asset pipeline—in short, a clean and neat way to provide assets. Assets are images, JavaScript, and CSS. Earlier, we had to put all the `.js`, `.css` and image files in the `public/` directory. The problem with this was that if a page did not want to use a particular JavaScript or a CSS file, it still loaded them all, although it was using the same layout (but without JavaScript or CSS).

All the custom JavaScript was put in an `application.js` JavaScript file and all custom CSS was put in a common CSS file. With the asset pipeline, it's a more streamlined and customized approach to serving assets. All the assets are compiled and compressed into a single JS and CSS file with an e-tag (an expiry tag).



Read more about sprockets and the asset pipeline at http://guides.rubyonrails.org/asset_pipeline.html. Sprockets is a gem that helps in assembling and compiling assets using directives.

Rails 3 projects are bundled with the `jquery-rails` gem and hence we have access to jQuery by default. We also want to use Twitter Bootstrap. Hence, we have bundled the `bootstrap-sass` gem in the Gemfile. To bundle all the Bootstrap JavaScript files in our asset pipeline, we use the Sprocket directive shown next. If we open the `app/assets/application.js` file, we would see the following:

```
//= require jquery  
//= require jquery_ujs  
//= require bootstrap  
//= require_tree .
```

This automatically includes all the bootstrap JavaScript into the asset pipeline. As we can see, we also include `jquery`, `jquery_ujs` and any custom JavaScript file in the `app/assets/javascripts` directory. This keeps our project code incredibly clean.

Just like we have included the Bootstrap JavaScript files, we also need to include the Bootstrap CSS files. In the `app/assets/stylesheets/application.css.sass`, the SASS file, we invoke the following command to include all the Bootstrap CSS styles:

```
@import 'bootstrap'
```

Designing the Authors listing page

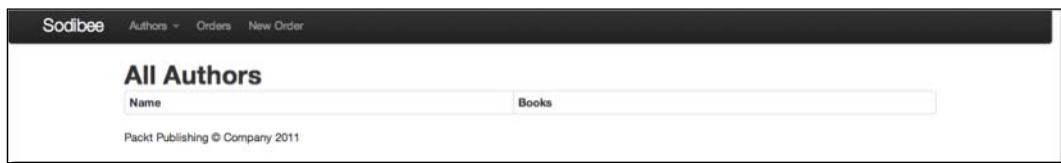
So, what and how do we render the authors? We want to list the author in a table along with their books!

Time for action – listing authors

Here is the `app/views/authors/index.html.haml`:

```
%h1 All Authors
%table{:class => "table table-striped table-bordered table-condensed"}
  %thead
    %tr
      %th Name
      %th Books
  %tbody
    - @authors.each do |author|
      %tr
        %th= link_to author.name, edit_author_path(author)
        %th= author.books.collect(&:title).to_sentence
```

Now when we invoke `http://localhost:3000/authors` via the browser, we should see the following screenshot:



As we have not added any authors yet, it's empty, but looking pretty! If you were using the same MongoDB database while experimenting during the earlier chapters, you would actually see the authors and their books here!

What just happened?

Before we see the View code in detail, let's quickly revisit our Controller code:

```
class AuthorsController < ApplicationController

  # GET /authors
  def index
    @authors = Author.all.includes(:books)
  end

  ...
end
```

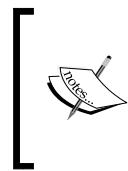
We are fetching all the authors and their books in the instance variable `@authors` (eager loading the books, remember?). Now let's see the View code in detail:

```
%h1 All Authors
%table{:class => "table table-striped table-bordered table-condensed"}
  %thead
    %tr
      %th Name
      %th Books
  %tbody
    - @authors.each do |author|
      %tr
        %th= link_to author.name, edit_author_path(author)
        %th= author.books.collect(&:title).to_sentence
```

The preceding part of the code creates the table. Notice, that we have given some styles to the table. These are picked up from the Bootstrap:

```
%h1 All Authors
%table{:class => "table table-striped table-bordered table-condensed"}
  %thead
    %tr
      %th Name
      %th Books
  %tbody
    - @authors.each do |author|
      %tr
        %th= link_to author.name, edit_author_path(author)
        %th= author.books.collect(&:title).to_sentence
```

What we just saw, is the core of the Haml logic and Ruby code integrated. We are iterating over the `@authors` array and listing the authors name in the first column. In the second column, we are collecting the titles of all the books of that author and converting them into a sentence—a little ActiveSupport magic here!



Read about Bootstrap at <http://twitter.github.com/bootstrap/>

ActiveSupport provides a lot of utility methods for Controllers and Views. Having a good knowledge of these methods can really help us write very very good code.

Let's get a little deeper into this particular Ruby code and understand some more facets of Ruby! Take a look at the following line of code:

```
author.books.collect(&:title).to_sentence
```

author is an Author object.

author.books is an array of books that this author has written.

collect is a method that iterates over an array and returns the objects that match the criteria in the block of code provided. The one we just saw is a concise code and this could also be written as follows:

```
author.books.collect do |book|
  book.title
end
```

The preceding code basically collects all the titles of the books. map is an alias of collect. Ruby has plenty of such alias methods to help programmers from different programming backgrounds to remember method names. collect has its roots from Smalltalk while map or transform is used in most other higher-level languages.

The to_sentence method is pretty interesting. ActiveSupport goes the distance to make our life easy with arrays! Let's see this using the following examples:

```
irb> [1, 2, 3].to_sentence
=> "1, 2, and 3"

irb> [1,2].to_sentence
=> "1 and 2"

irb> [].to_sentence
=> ""

irb> [1].to_sentence
=> "1"

irb> [1, 2, 3, 4].to_sentence
=> "1, 2, 3, and 4"
```

Isn't that beautiful? `to_sentence` automatically manages punctuations and the last "and"! If we add authors and books, we should see something, as shown in the following screenshot:

The screenshot shows a web application interface titled "Sodibee". At the top, there are navigation links: "Authors" (with a dropdown arrow), "Orders", and "New Order". Below this, a title "All Authors" is displayed above a table. The table has two columns: "Name" and "Books". The data in the table is as follows:

Name	Books
Charles Dickens	A tale of Two Cities, Count of Monte Cristo, and Oliver Twist
Mark Twain	Adventures of Tom Sawyer
Gautam Rege	Ruby And MongoDB Guide
Alexandre Dumas	

At the bottom of the page, a copyright notice reads "Packt Publishing © Company 2011".

Adding new authors and their books

When we create authors, we want their books to be added too at that time. In other words, we want the form for creating a book to be nested inside the form for creating an author. These are called nested attributes. First we need to tweak the `Author` model for this.

Time for action – adding new authors and books

First let's see how the `Author` model has changed a bit to accommodate book attributes! Have a look at the following code:

```
class Author
  include Mongoid::Document

  field :name, type: String

  validates_presence_of :name

  has_one :address, as: :location, autosave: true, dependent: :destroy
  has_many :books, autosave: true, dependent: :destroy

  accepts_nested_attributes_for :books, :address, allow_destroy: true
end
```

Now we add the nested template `app/views/authors/new.html.haml`, HAML file:

```
%h2 New Author

= simple_nested_form_for(@author, :html => { :class => 'well form-
horizontal' }) do |f|
  = f.input :name
  = render 'shared/address', :f => f
```

```
%h2 Books
= f.fields_for :books do |b|
  %fieldset{:class => 'well'}
    = b.input :title
    = b.input :publisher
    = b.association :categories, collection: Category.all
    = b.link_to_remove "Remove", :class => 'btn btn-danger btn-mini'

  = f.link_to_add "Add Book", :books, :class => 'btn btn-success'
= f.submit :class => 'btn-primary'
```

The preceding code is the template that will be rendered when the AuthorsController#new action is invoked from the URL `http://localhost:3000/authors/new`, that is, when we click on **New Author** from the menu bar we will see the following screen:

The screenshot shows a web application interface for adding a new author. At the top, there's a navigation bar with links for 'Authors', 'Orders', and 'New Order'. The main content area is titled 'New Author'. It contains two sections: 'Author Information' and 'Books'. In the 'Author Information' section, there are fields for 'Name' (filled with 'Mark Twain'), 'Street', 'Zip', and 'State'. In the 'Books' section, there's a table-like structure with columns for 'Title' (filled with 'Oliver Twist'), 'Publisher' (filled with 'Penguin'), and 'Categories' (containing 'Drama' and 'Fiction'). A red 'Remove' button is located below the categories. At the bottom of the form are two buttons: a green 'Add Book' button and a blue 'Create Author' button.

What just happened?

A lot just happened! Let's take it step by step. Remember we have installed `simple_form` and `nested_form` gems! These kick in here and do their magic. Let's see the code of nested attributes first:

```
class Author
  include Mongoid::Document
```

```
field :name, type: String

validates_presence_of :name

has_one :address, as: :location, autosave: true, dependent: :destroy
has_many :books, autosave: true, dependent: :destroy

accepts_nested_attributes_for :books, :address, allow_destroy: true
end
```

The `accepts_nested_attributes_for` method ensures that for the `Author` object, it will also directly access or save its books and address. We have seen the code in the Controller already where the address and book objects are built! Here is a brief reminder:

```
def new
  @author = Author.new
  @author.build_address
  @author.books.build
end
```

Now, we shall see the code of the View:

```
%h2 New Author

= simple_nested_form_for(@author, :html => { :class => 'well form-horizontal' }) do |f|
  = f.input :name
  = render 'shared/address', :f => f

%h2 Books
= f.fields_for :books do |b|
  %fieldset{:class => 'well'}
    = b.input :title
    = b.input :publisher
    = b.association :categories, collection: Category.all
    = b.link_to_remove "Remove", :class => 'btn btn-danger btn-mini'

  = f.link_to_add "Add Book", :books, :class => 'btn btn-success'

  = f.submit :class => 'btn-primary'
```

Using `simple_nested_form_for` instead of the traditional `form_for` gems makes the form alive to nested fields as well as `simple_form` fields!

 **Configuring for nested_form**

When using nested form, we initially need to add a custom JavaScript file. This is done using the `rails generate nested_form:install` command.

This command generates a `public/javascripts/nested_form.js` file. It is recommended that this be moved to `app/assets/javascripts` directory so that it gets bundled in the asset pipeline.

Have a look at the following code snippet:

```
%h2 New Author

= simple_nested_form_for(@author, :html => { :class => 'well form-horizontal' }) do |f|
  = f.input :name
  = render 'shared/address', :f => f

%h2 Books
= f.fields_for :books do |b|
  %fieldset{:class => 'well'}
    = b.input :title
    = b.input :publisher
    = b.association :categories, collection: Category.all
    = b.link_to_remove "Remove", :class => 'btn btn-danger btn-mini'

  = f.link_to_add "Add Book", :books, :class => 'btn btn-success'

  = f.submit :class => 'btn-primary'
```

This is `nested_form` kicking in!

 `simple_form` methods set the form fields based on the type of data, so it will automatically render the string as a text field, a date as the default date format fields, and so on.

It also creates a `<label>` field based on the name of the field.

If that was not enough, it also checks on validations and if a field has `:presence => true` (for example, the `:name` field of `Author`), it will automatically add a `*` to the label and a `required = "required"` to the form input element.

When using `simple_nested_form_for`, the `fields_for` picks up the association (remember an author has many books) and renders the book object fields.

`simple_form` also understands these associations automagically! As books and categories have a many-to-many relation, it shows the categories as a multi-select input!

We can add more books using the **Add Book** button and remove book objects via the **Remove** button.



`nested_form` uses a combination of JavaScript and a blueprint template that is generated using the association and the fields of the associated object



Address and **Books** are now populated as nested attributes:

The screenshot shows a web application interface for creating a new author. At the top, there's a navigation bar with links for 'Sodibee', 'Authors', 'Orders', and 'New Order'. Below that, the main title is 'New Author'. The form has two main sections: 'Address' and 'Books'. The 'Address' section is highlighted with a red box and contains fields for 'Name' (Mark Twain), 'Street', 'Zip', and 'State'. The 'Books' section is also highlighted with a red box and contains fields for 'Title' (Oliver Twist), 'Publisher' (Penguin), and 'Categories' (with 'Drama' and 'Fiction' selected). At the bottom of the form, there are buttons for 'Add Book', 'Create Author', and 'Remove'.

Similarly, we can add and remove books using the `nested_form` helpers. Nested form enables some smart ways to add more books and remove them using some simple JavaScript and blueprint templates. A **blueprint template** is an HTML `<div>` tag that is not rendered, but used for creating more `<div>` tags which are part of the form that would be sent to the server for creation of the author and the author's books:

The screenshot shows a 'Books' management interface. It displays two entries. The first entry has a filled-in title ('Oliver Twist'), publisher ('Penguin'), and categories ('Drama', 'Fiction'). A 'Remove' button is highlighted with a red box. The second entry has empty fields for title, publisher, and categories, with a 'Remove' button. At the bottom, there are 'Add Book' and 'Create Author' buttons.

But that's not all! `simple_form` also helps us render validations and errors properly! Remember that the title of the book and the name of the author are mandatory, these are shown with an asterisk next to the label!

What if the form is submitted but has some validation errors? We know that the `new` action is rendered and the `@author` object has `errors` populated. But how are they shown? They are shown as follows:

The screenshot shows a 'New Author' form with validation errors. The 'Name' field is marked with an asterisk and has an error message 'can't be blank'. The 'Books' section shows a validation error for the 'Title' field, which is marked with an asterisk and has the message 'can't be blank'.

Welcome to Rails!

Have a go hero

- ◆ Why don't you Bootstrap the members or the orders MVC?
- ◆ Why don't you implement the Author Edit functionality?
- ◆ Members have an address (it's polymorphic)
- ◆ Orders have an embedded type, Purchase or Lease.
- ◆ Books can have reviews and votes from members (nested attributes!)

The Sinatra way

Now that we have seen this the Rails way, let's see how this is done using Sinatra and Rack!

Time for action – setting up Sinatra and Rack

As we have seen before, Sinatra requires very little configuration. Here is our Gemfile:

```
source 'https://rubygems.org'

gem 'sinatra'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'
gem 'mongoid'
gem 'bson'

gem 'haml'
```

We have removed a lot of gems (such as `rails`, `simple_form`, `nested_form`, `bootstrap-sass`, and all the asset gems). This is because some are very Rails dependent. To get the power of Bootstrap JavaScript and the CSS, we simply copy them in a directory where we shall keep all the static assets:

```
$ ls -R public/
css/    img/    js/

public//css:
bootstrap.css

public//img:
```

`glyphicons-halflings-white.png` `glyphicons-halflings.png`

```
public//js:  
bootstrap.js      jquery.js
```

Now, we configure Sinatra to "talk" to MongoDB! This is done as follows:

```
require 'mongoid'  
require 'sinatra'  
  
configure do  
  Mongoid.configure do |config|  
    name = "sodibee_development"  
    host = "localhost"  
    config.master = Mongo::Connection.new.db(name)  
    config.persist_in_safe_mode = false  
  end  
end
```

The MongoDB models don't change at all. And as the core of the application is in these models, this makes life really easy! All we have to do is load the Ruby classes! This is done as follows:

```
require 'mongoid'  
require 'sinatra'  
  
configure do  
  Mongoid.configure do |config|  
    name = "sodibee_development"  
    host = "localhost"  
    config.master = Mongo::Connection.new.db(name)  
    config.persist_in_safe_mode = false  
  end  
  
  enable :sessions  
end
```

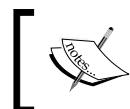
Routes and Controller logic is bundled up together in Sinatra! So, we can simply take some Controller logic out of the Rails application and put it in our `app.rb` file, as shown in the following code:

```
get "/authors" do  
  @authors = Author.all  
  haml :'authors/index'  
end
```

Here is what our layout looks like. This is the `views/layout.haml`—the default layout:

```
!!!
%html{:lang => :en}
  %head
    %meta{:charset => "utf-8"/>
    %title Sodibee Library Manager
    %script{:src => "/js/jquery.js", :type => "text/javascript"}
    %script{:src => "/js/bootstrap.js", :type => "text/javascript"}
    %script{:src => "/js/bootstrap-dropdown.js", :type => "text/
javascript"}
    %script{:src => "/js/bootstrap-collapse.js", :type => "text/
javascript"}
    %link{:href => '/css/bootstrap.css', :rel => 'stylesheet', :type
=> 'text/css'}
```

```
%body
  .navbar
    .navbar-inner
      .container-fluid
        %a{:href => "/", :class => 'brand'} Sodibee
        %ul.nav
          %li.dropdown
            %a{:class => 'dropdown-toggle', :href => '#', :data =>
{:toggle => 'dropdown'}}
              = "Authors"
            %b.caret
            %ul.dropdown-menu
              %li
                %a{:href => '/authors'} List Authors
              %li
                %a{:href => '/authors/new'} New Author
            %li
              %a{:href => "/orders"} Orders
            %li
              %a{:href => "/orders/new"} New Order
  .container
    .content
      = yield
  .footer
    %p Packt Publishing &copy; Company 2011
```

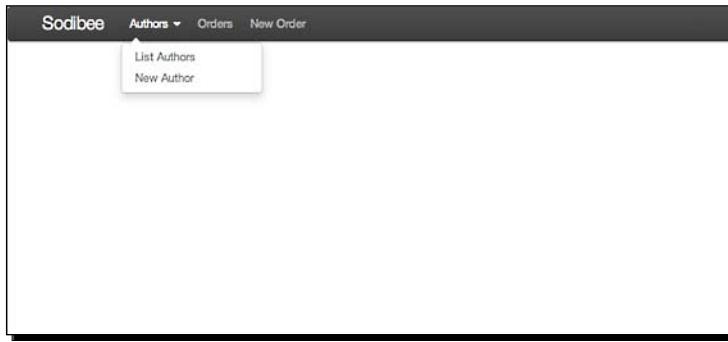


As this is not Rails, there is no `ActionView` and its `FormHelpers` available. So, we need to rewrite the Views and make them independent of Rails. This increases our overhead a little.

Let's rackup and be on our way! Let's execute the following commands:

```
$ rackup config.ru
INFO  WEBrick 1.3.1
INFO  ruby 1.9.2 (2011-07-09) [i386-darwin9.8.0]
INFO  WEBrick::HTTPServer#start: pid=17348 port=9292
```

The result is visible! The browser will display our application as follows:



What just happened?

We successfully set up a Sinatra application with Rack and MongoDB! And as we have seen, it isn't very difficult to move our code between compliant Rack applications! Points to note are as follows:

- ◆ The MongoDB models (the core) do not change at all
- ◆ The Controller code remains the same
- ◆ The routes are configured in a slightly different way in Sinatra and Rails
- ◆ We need to make a lot of changes in the Views because in Rails, we used FormHelpers and ActionView methods that are not available with Sinatra

Have a go hero

Why don't you try and add the /authors/new functionality?

Testing and automation using RSpec

No application is complete without proper tests in place. We shall not go into a lot of automated testing concepts here because there are books about this. We shall touch upon a few concepts though.

Understanding RSpec

RSpec is a popular autotest tool used very heavily, especially in a Rails application. We can test Models, routes, Controllers and even Views in an automated way.

Time for action – installing RSpec

Ensure that you have the following gem in your Gemfile:

```
group :development, :test do
  gem 'rspec-rails'
  gem 'spork'
  gem 'faker'
end
```

In our Rails application, to set up RSpec we need to invoke the following command:

```
$ rails generate rspec:install
  create .rspec
  create spec
  create spec/spec_helper.rb
```

Removing specific ActiveRecord configuration.

You will need to comment the following lines in the `spec/spec_helper.rb` file to ensure there aren't any errors due to ActiveRecord:

- ◆ `config.fixture_path = "#{::Rails.root}/spec/fixtures"`
- ◆ `config.use_transactional_fixtures = true`

Now, we can write some RSpec code on our own. We can write the `Author` model test specifications in `spec/models/author_spec.rb`:

```
require 'spec_helper'

describe Author do
  it "should be created if name is provided" do
    Author.create(name: "test").should be_valid
  end

  it "should not be created without a name" do
    Author.create.should_not be_valid
  end
end
```

To see if the test cases pass, we can run RSpec, as follows:

```
$ rspec spec/models  
..  
  
Finished in 5.08 seconds  
2 examples, 0 failures
```



Depending on the machine, the Ruby version, the Rails version, and the RSpec version, the speed of the tests may vary.



What just happened?

Let's look at what we tested! But first, let's look at some basics of RSpec:

- ◆ `describe`: This is a method (yes, a method) that takes a string and a block of code which has all the test cases in it.
- ◆ `it`: This is another method that takes a string as the name of the test case and a block of code for the actual test case.
- ◆ `should`: This is a method that does the actual validation of the test case. If this method returns `true`, the test case passes, otherwise it fails.
- ◆ `should_not`: This is the inverse of the `should` method.
- ◆ `be_valid`: This is a method which validates an object's existence.

There are plenty of other methods that you can read up in the RSpec book! Let's look at one test case! Have a look at the following code snippet:

```
it "should be created if name is provided" do  
  Author.create(name: "test").should be_valid  
end
```

Here, we create an author and test if it "should be valid". If the object is successfully created, it will not be `nil` or in other words, it will be valid!

Notice that running two tests took about five seconds! Welcome spork— a speedy way to get RSpec up and running.

Time for action – sporking it

First, install spork – add it to the Gemfile if it's not already there in the following manner:

```
gem 'spork'
```

Now, we install spork in the following manner:

```
$ spork --bootstrap
Using RSpec
Bootstrapping /Users/gautam/Documents/books/ruby_and_mongodb/Book/code/
sodibee/spec/spec_helper.rb.

Done. Edit /Users/gautam/Documents/books/ruby_and_mongodb/Book/code/
sodibee/spec/spec_helper.rb now with your favorite text editor and follow
the instructions.
```

Now, if we do indeed follow the instructions, we can configure spork. Open the `spec/spec_helper.rb` and move the original `spec_helper` code inside the `prefork` code. This will preconfigure spork for RSpec! This is what the file looks like:

```
require 'spork'

Spork.prefork do
  ENV["RAILS_ENV"] ||= 'test'
  require File.expand_path("../config/environment", __FILE__)
  require 'rspec/rails'
  require 'rspec/autorun'

  Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

  RSpec.configure do |config|
    config.infer_base_class_for_anonymous_controllers = false
  end

end

Spork.each_run do
  # This code will be run each time you run your specs.

end
```

Now, let's see what changes. First, start spork in one terminal, as follows:

```
$ spork
Using RSpec
Preloading Rails environment
```

```
Loading Spork.prefork block...
Spork is ready and listening on 8989!
```

Now, in another terminal let's run RSpec and see what happens:

```
$ rspec spec
..
Finished in 0.04797 seconds
2 examples, 0 failures
```

What just happened?

Wow! We finished the test cases in 0.04797 seconds instead of the earlier run of 5.08 seconds! That's a huge boost to testing. What spork does is that it preloads the Rails environment and runs all the test cases in parallel.

Have a go hero

Let's write out test cases for books, orders and members!

Documenting code using YARD

Just like testing is very important, so is documentation. After some research, I strongly recommend using YARD. YARD generates HTML documentation for models and Controllers.

You can install YARD using the following command:

```
$ gem install yard
```

To write code documentation, this is how our file would look. I am taking the example of the Book model. This is what it looks like:

```
##
# This class defines the details of a Book.
#
class Book
  include Mongoid::Document

  # @return [String] The title of the book
  field :title, type: String

  # @return [String] The publisher of the book
  field :publisher, type: String
```

```
# @return [String] The date the book is published on
field :published_on, type: Date

# @return [String] The price of the book is a localized string
#           Depending on the locale, the prices are updated as
#           per their currency rate.
field :price, localize: true

# @return [Array] An array of votes in the format that we can
identify
#           upvotes and downvotes! Hence each element of the array
#           is an hash in a fixed format.
#           { 'name' => 1 } # => upvote
#           { 'name' => -1 } # => downvote
field :votes, type: Array

# @return [Author] This is the author of the book.
belongs_to :author

# @return [Array] The array of Category objects.
#           These are the categories that this book belongs
to.
has_and_belongs_to_many :categories

# @return [Array] This returns the array of all embedded reviews.
embeds_many :reviews

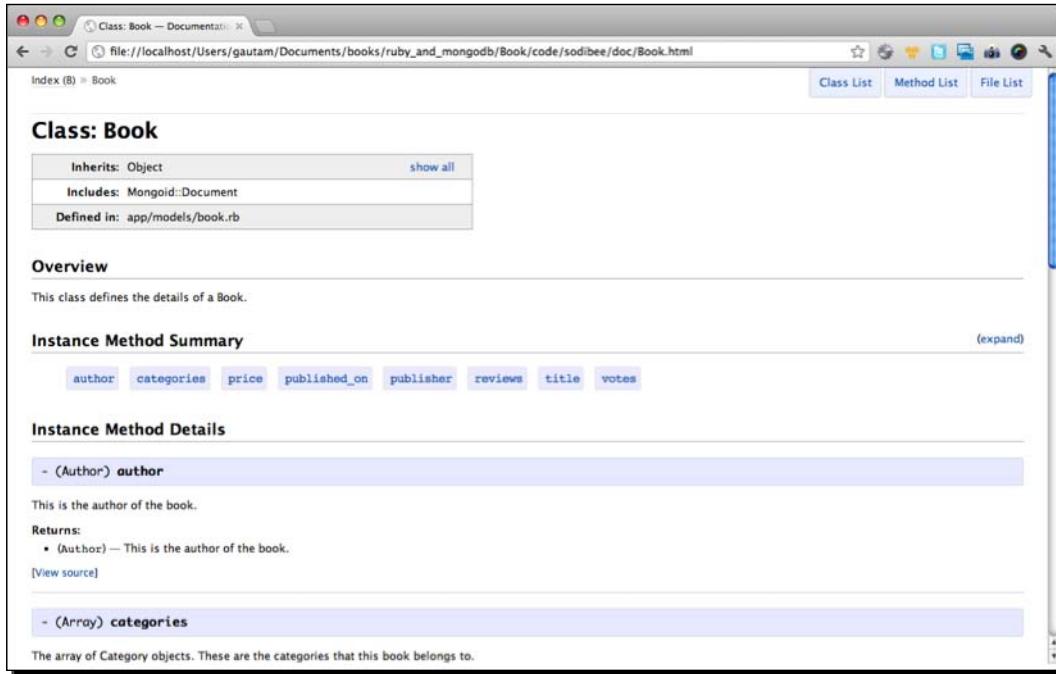
# @return [Boolean] true if the validation of title passes
validates :title, presence: true

end
```

To generate the documentation, issue the following command:

```
$ yard doc
Files:          11
Modules:        1 (    1 undocumented)
Classes:        10 (    8 undocumented)
Constants:      0 (    0 undocumented)
Methods:        5 (    0 undocumented)
43.75% documented
```

This generates the documentation, as shown in the following screenshot:



YARD documentation is all in markdown. And it supports special tags such as @params, @return that enable us to write easy and good documentation. Go ahead and learn it!

Pop quiz – it's all about the web

1. Is it true that Rails and Sinatra are Rack applications?
 - a. Yes.
 - b. No.
 - c. Rails can be configured to not use the Rack.
 - d. What is the rack again?
2. How is data made available to the Views from the Controllers?
 - a. No data from the Controllers is available for the Views.
 - b. All instance variables in the Controllers are available to the Views.
 - c. All local variables in the Controllers are available to the Views.
 - d. JSON data is passed to the Views.

3. What does `accept_nested_attributes_for` do?
 - a. It accepts nested attributes for an HTTP request.
 - b. A parent model can access the data of child objects using this method.
 - c. It's a method that enables child objects to be created or updated, along with a parent object creation or update.
 - d. It nests or embeds child objects into the parent.
4. Which of the following enables us to write HTML templates with embedded Ruby in it?
 - a. Sass.
 - b. Bootstrap.
 - c. CoffeeScript.
 - d. Haml.
5. Which of the following is not true for the Rails asset pipeline?
 - a. It compresses assets like JavaScript, Images and CSS for speed.
 - b. It can process Sass and CoffeeScript and compile them into CSS and JavaScript.
 - c. It uses the sprocket gem for managing the asset pipeline.
 - d. It compiles Ruby code into HTML.

Summary

WOOt! This has been a chapter where we actually built a fully functional web application using Rails and Sinatra. We have seen how to model a web application in the previous chapters. Now, we used them. We saw what Rails routes are and how they are processed. We were introduced to Twitter Bootstrap, Haml and Sass. We also looked at some very useful gems such as, `simple_form` and `nested_form`. We briefly looked at how to test an application and even document it!

You're all set to explore the wonderful world of MongoDB and Ruby now. The more you experiment the more you will learn. The next couple of chapters would deal with leveraging MongoDB specific features. In the next chapter, we shall leverage MongoDB geospatial indexing to make our applications location aware. The last chapter deals with scaling MongoDB and some more Map/Reduce!

9

Going Everywhere – Geospatial Indexing with MongoDB

MongoDB has geospatial indexing enabled by default. Woh! Let's talk in normal English here.

This is the age of location sensitive information. If I am in London, I would like to know the local news, deals, restaurants, and maybe even friends who are nearby. There are services that do this already – Foursquare, Gowalla (now with Facebook), Google Maps, and now Facebook.

The basic concept of geolocation is to isolate the exact location (to as close as possible) and provide services related to that location. Geospatial indexing is a way to use this information from the database. We index these coordinates because it helps us query faster.

So, how is this related to MongoDB? Remember that, when we say "near a location", it could mean a circle, a rectangle or even a sphere around our location! The distance could be in miles or kilometers or meters. This causes a sizable amount of complexity in calculation. We have to find out the range of nearby coordinates and then look up the database for information that is within that range! Not an easy task, as we shall soon see.

MongoDB comes to our rescue because it already has the capability of querying, storing coordinates and looking up geolocation data.

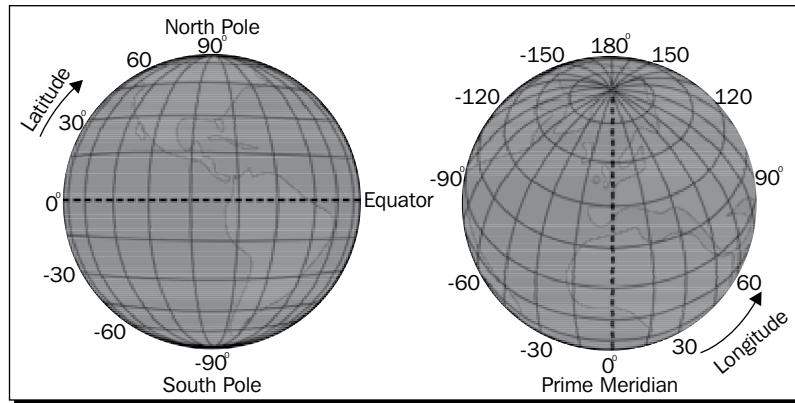
In this chapter we shall learn the following:

- ◆ What do we mean by geolocation
- ◆ How is the geolocation calculated
- ◆ How can we store this information in MongoDB
- ◆ How can we use it in our application via Mongoid

Geographical Information Systems(GIS) are all based on geolocations. Some relational databases do support geospatial indexing, for example PostGIS, which is an extension to PostgreSQL. MongoDB has these capabilities built right into it.

What is geolocation

Let's split the word geolocation. **Geo** means the earth and **location** means position. So geolocation means our position on the earth. As we know, the earth is divided into latitudes and longitudes, as shown in the following image taken from Wikipedia:



As we can see, latitudes range from 90° to -90° and longitudes range from 180° to 0° . The 0° latitude is the equator and the 0° longitude runs via Greenwich in UK. If we see both, the latitudes and longitudes, the earth is entirely divided into segments and we can identify every position on the earth's surface.

At the equator, the distance between the degrees in the longitudes is approximately 111.3 km and this distance keeps reducing as the latitude goes North or South. At 60° latitude, the distance between the degrees in the longitudes is 55.65 km.

How accurate is a geolocation

Understandably, we need to know both, the latitude and the longitude to identify the location. But the distance between latitudes and longitudes is too large to get the exact location, say within a few meters!

To cater to this, the distance between each degree of latitude and longitude is divided into 60 minutes and each minute is divided into 60 seconds. Doing this gets us even closer to pinpointing a location. Keep in mind that the distances between each longitude and latitude vary for every second! At the equator (0° latitude), one-second difference between the latitudes is about 30.715 m and decreases as we move towards the poles. One-second difference between longitudes at the equator is 30.92 m and one-second difference between longitudes at 30° latitude is 26.76 m.

Given that the earth's radius is about 6.3 million meters (6371 km as per MongoDB), getting an accuracy of within 30 m suits us just fine. Generalizing this, for a 0.0001° change, the accuracy is between 5 m and 11 m!



The earth's radius has been calculated using various different models.

Mean radius: 6,371.009 km.

Great-circle radius: 6,372.797 km.

Authalic radius: 6,371.0072 km.

Volumetric radius: 6,371.0008 km.

Meridional radius: 6,367.445 km.

Read more at http://en.wikipedia.org/wiki/Earth_radius#Mean_radii.

Converting geolocation to geocoded coordinates

Typically a position on the earth is written as 40°26' 21"N 79°58' 36"W. This means the latitude is 40 degrees north latitude and a further 26 minutes and 21 seconds northward and 79° west longitude and a further 58 minutes and 36 seconds westward!

Using this convention is easy to read but very difficult for calculations. So, we convert these **Degrees Minutes Seconds (DMS)** to a Decimal degree. Basically, we convert the minute and second to a fraction. Simply put, there are 3600 seconds between degrees. So, 1 second is approximately 0.00027777 minutes. In the previous example, 26 minutes and 21 seconds is $(26 * 60) + 21 = 1,581$ seconds.

So, the Decimal degree of latitude 40°26' 21" N is 40.4390437. North is a positive result and south is a negative result. Similarly, east is a positive result and west is a negative result. It is these Decimal degrees that we save as float values in the MongoDB that act as the coordinates!

Identifying the exact geolocation

Converting geolocation to geocoded coordinates is one thing but how does one find the actual location on the earth? Am I sitting in the Sahara desert in Africa or in a pub in London or at home in India? There are various techniques and tools that help us find out this information:

- ◆ **GPS devices.** These use the geostationary satellites for isolating the exact coordinates of the device and in turn your exact location. These are by far the most accurate. These are used heavily in navigation systems.



Most modern devices (such as, smartphones and tablets) support GPS. Access to GPS satellites has traditionally been under the gamut of the military and only in the last decade has GPS access been provided for commercial use by navigation systems.

- ◆ **Mobile phone.** Depending on the phone, we can get the coordinates in varying levels of accuracy. Some smart phones (such as, iPhone, BlackBerry, and Android) use advanced location-based applications that need to be installed. Some phones also use a hybrid way (a combination of network-based and handset-based positioning) to find the exact location.
- ◆ **Mobile Network.** The mobile network operators get geolocation information from the location of the cell-phone tower. This is not very accurate for identifying the exact location but for handsets that do not have any software installed, this serves well. Some SIM cards too can be used for getting the exact location using raw radio measurements from the handset.
- ◆ **Network devices.** When we are connected to the Internet, our devices (such as, phones or computers) are assigned an IP address. This is the least accurate means of getting a geolocation, but the router static IP address can also give us a geolocation. This depends on various **Internet Service Providers** (ISP), the geography, Internet density, and so on.
- ◆ **Map APIs.** Google, Yahoo!, geocoder, and Bing are some services which have latitudes and longitudes mapped to addresses in the world. They are by no means complete but they are very extensive and ever increasing. These Map APIs are very heavily used in web applications to find the exact latitude and longitude of an address.



HTML 5 provides support to find the geolocation of the machine using one or more of the ways mentioned in the preceding list. Read more at <http://dev.w3.org/geo/api/spec-source.html>.

In a nutshell, it's almost always possible to get some sort of a geolocation but with varying levels of accuracy.

 It may be worth our time to see the future of geolocation-sensitive applications!

Foursquare, Gowalla (now with Facebook), Yelp, Twitter, and a lot of other social media applications are using location-based applications for generating revenue. This has lead to a new era of "Social Location Marketing" (Do read *Social Location Marketing: Outshining Your Competitors on Foursquare, Gowalla, Yelp & Other Location Sharing Sites* by Simon Salt).

There are a lot of web portals that target the local communities for getting good local deals, local news, promoting local events and even local organizations! This causes the web portal to give us more relevant information and thereby engages users. This, in turn increases revenues and profit.

Storing coordinates in MongoDB

Let's see how we can add geospatial indexes to MongoDB.

Time for action – geocoding the Address model

As the Address is a model for storing the location, we can use it for geospatial indexing! This is done as follows:

```
class Address
  include Mongoid::Document

  field :street, type: String
  field :zip, type: Integer
  field :city, type: String
  field :state, type: String
  field :country, type: String

  field :coordinates, type: Array
  index [[ :coordinates, Mongo::GEO2D ]]

  belongs_to :location, polymorphic: true
end
```

The indexes need to be created in the model manually. Mongoid will not issue commands to create them unless explicitly told to do so. Let's create indexes as follows:

```
$ rake db:mongoid:create_indexes

Generated indexes for Address
Generated indexes for Author
Generated indexes for Book
Generated indexes for Category
Not a Mongoid parent model: app/models/lease.rb
Generated indexes for Member
Generated indexes for Order
Not a Mongoid parent model: app/models/purchase.rb
Not a Mongoid parent model: app/models/review.rb
```

What just happened?

MongoDB has now created indexes for the models.



Index creation is not geospatial specific. We could use this command for all models too. Notice that it has created indexes for all models. Indexing helps in speeding up queries.



Have a look at the following code snippet:

```
class Address
  include Mongoid::Document

  field :street, type: String
  field :zip, type: Integer
  field :city, type: String
  field :state, type: String
  field :country, type: String

  field :coordinates, type: Array
  index [[ :coordinates, Mongo::GEO2D ]]

  belongs_to :location, polymorphic: true
end
```

Here we are creating a standard array but we shall ensure that it stores only two values, the latitude first and then the longitude. For example, [10.123244, -87.783562]. The index actually tells MongoDB that this is a Mongo::GEO2D index. It also sets the default minimum and maximum value to -180 to 180 (that is, the range of decimal degrees). We can override this range if we want, as follows:

```
index [ [:coordinates, Mongo::GEO2D] ], min: -500, max: 500
```

Internally, it sets the index as a 2d index. 2d means two dimensional that is, it knows that it is a spatial index. When we issue the command to create indexes, Mongoid creates indexes by default for the `_id` field, that is, the object ID. It also created a 2d index for addresses. This can be seen on the MongoDB console:

```
Fri Mar 16 14:40:30 [conn262] query sodibee_development.system.namespaces
nscanned:25 nreturned:25 reslen:1556 228ms
Fri Mar 16 14:40:30 [conn262] build index sodibee_development.addresses {
coordinates: "2d" }
Fri Mar 16 14:40:30 [conn262] build index done 3 records 0.3 secs
Fri Mar 16 14:40:30 [conn262] insert sodibee_development.system.indexes
620ms
```

It's also interesting to note that embedded documents, such as `Lease`, `Purchase`, and `Review` do not get indexed on their `_id` fields because they cannot be directly accessed. However, you can index fields inside embedded documents using the dot notation! If we require to say the `:price` from the `Purchase` model we can index it too! This can be done as follows:

```
class Order
  ...
  embeds_one :purchase

  index :"purchase.price"
end
```

Testing geolocation storage

Ok! Back to geospatial indexing. Suppose our latitude and longitude of an address is known (we shall see soon, how we can determine it programmatically), we can add it to the database.

Time for action – saving geolocation coordinates

Suppose our latitude and longitude is 10.123123 and -87.1231231 respectively, we can add it directly to the coordinates array, as:

```
irb> a = Author.last
=> #<Author _id: 4f55abf8fed0eb2f6c00002d, _type: "Author", name:
"Gautam Rege">

irb> a.address
=> #<Address _id: 4f55abf8fed0eb2f6c00002e, _type: "Address", street:
"101 Union Street", zip: nil, city: "Pasedena", state: "CA", country:
"US", coordinates: nil, location_type: "Author", location_id: BSON::Objec
tId('4f55abf8fed0eb2f6c00002d')>
```

```
irb> a.address.coordinates = [ 10.123123, -87.1231231 ]
=> [10.123123, -87.1231231]

irb> a.save
=> true
```

What just happened?

We save the coordinates into the array.



So, how did one get the latitude and longitude anyway?

Using Map APIs from Google (or Yahoo!, Bing and geocoder), we can get the latitude and longitude of a particular address if Google Maps can find that address. This is called geocoding. In Ruby, we have plenty of gems available for this. I personally recommend geocoder for this.

Using geocoder to update coordinates

We can use the geocoder gem to find the latitude and longitude of some actual address.

Time for action – using geocoder for storing coordinates

Add geocoder to the Gemfile first:

```
gem 'geocoder'
```

Now let's update the Address model, as follows:

```
class Address
  include Mongoid::Document
  include Geocoder::Model::Mongoid

  field :street, type: String
  field :zip, type: Integer
  field :city, type: String
  field :state, type: String
  field :country, type: String
  field :coordinates, type: Array

  belongs_to :location, polymorphic: true

  geocoded_by :formatted_addr
  after_validation :geocode
```

```

def formatted_addr
  [street, city, state, country].join(',')
end

end

```

Now let's save some addresses. Execute the following commands:

```

irb> a = Author.new(name: "Gautam Rege")
=> #<Author _id: 4fbf4c78fed0ebcdd0000004, _type: "Author", name:
"Gautam Rege">

irb > a.address = Address.new(street: "102 Union Street", city:
"Pasedena", state: "CA", country: "US")
=> #<Address _id: 4fbf4caffed0ebcdd0000006, _type: "Address", street:
"102 Union Street", zip: nil, city: "Pasedena", state: "CA", country:
"US", coordinates: nil, location_type: "Author", location_id: BSON::Objec
tId('4fbf4c78fed0ebcdd0000004')>

irb> a.save
=> true

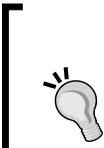
irb> a.address
=> #<Address _id: 4fbf4caffed0ebcdd0000006, _type: "Address", street:
"102 Union Street", zip: nil, city: "Pasedena", state: "CA", country:
"US", coordinates: [-118.1481163, 34.1467468], location_type: "Author",
location_id: BSON::ObjectId('4fbf4c78fed0ebcdd0000004')>

irb> a.address.coordinates
=> [-118.1481163, 34.1467468]

```

What just happened?

When we use geocoder gem, we have set up an `after_validation` callback. When the object is validated, we look up the geocoder, fetch its coordinates and save them in the object.



The geocoder gem has various lookup services that it can refer to, such as Google Map APIs, Yahoo! Maps, Bing, FreeGeoIP, among others and it defaults to Google – you can configure these lookups yourself.

Suppose you enter an unknown address and the service cannot find the geolocation, it returns and does not update the coordinates-you're on your own then!

Firing geolocation queries

Now that we have added the coordinates, let's see if this works!

Time for action – finding nearby addresses

Let's see if we can find addresses near some particular coordinates! Let's execute the following commands:

```
> Address.near(:coordinates => [10.123122, -87.1231230]).first  
=> #<Address _id: 4f55abf8fed0eb2f6c00002e, _type: "Address", street:  
"101 Union Street", zip: nil, city: "Pasedena", state: "CA", country:  
"US", coordinates: [10.123123, -87.1231231], location_type: "Author",  
location_id: BSON::ObjectId('4f55abf8fed0eb2f6c00002d')>
```

Wow!

What just happened?

When we search for data near some coordinates, it returns us the address we had. So far so good! Let's look at this particular statement of code:

```
> Address.near(:coordinates => [10.123122, -87.1231230]).first
```

Here `near` is a criterion that is available only for 2d indexes.

But wait, we did not specify how near or how far from the coordinates we should lookup, did we? Let's try something here. Let's see if `near` has a default nearby distance. If we search for `[0, 0]`, would this object be returned? Try executing the following command:

```
> Address.near(:coordinates => [0, 0]).first  
=> #<Address _id: 4f55abf8fed0eb2f6c00002e, _type: "Address", street:  
"101 Union Street", zip: nil, city: "Pasedena", state: "CA", country:  
"US", coordinates: [10.123123, -87.1231231], location_type: "Author",  
location_id: BSON::ObjectId('4f55abf8fed0eb2f6c00002d')>
```

Holy cow! What's going on here? By no means can `[10.123123, -87.1231231]` be anywhere near `[0, 0]`. Let's see what the mongo console says. Is this a bug in Mongoid, MongoDB, or are we doing something wrong? Let's see! Let's execute the following commands:

```
$ mongo
```

```
MongoDB shell version: 2.0.2  
useconnecting to: test
```

```
> use sodibee_development
switched to db sodibee_development

> db.addresses.find({ coordinates: { $near: [0, 0] } })
{ "_id" : ObjectId("4f55abf8fed0eb2f6c00002e"), "_type" : "Address",
"coordinates" : [ 10.123123, -87.1231231 ], "location_id" : ObjectId("4
f55abf8fed0eb2f6c00002d"), "location_type" : "Author", "state" : "CA",
"street" : "101 Union Street", "zip" : nil }
```

Woh! Here is how this works! "near" is a relative term, we have not told MongoDB what near is! So, MongoDB gets us the nearest 100 objects by default. As there is only one object in the Address collection, it gets returned. If we require to really get nearby objects within a particular range, we need to specify it using \$maxDistance.



\$maxDistance is always specified in radians. Converting to radians is trivial. MongoDB takes the earth's radius as 6371 km. So, if we want a range of 1000 km, it means it's $(1000 / 6371)$ radians that is, 0.1569 radians. Similarly, we can use any unit of distance and calculate the radians!

Now let's try this again:

```
> db.addresses.find({ coordinates: { $near: [0, 0] }, $maxDistance: 1 })
>
```

And we get an empty result, phew!

Now let's test these constraints with the coordinates [10.123123, -87.1231231]. We shall keep the latitude as 10° and change the longitude by 1° in both directions. Let's execute the following queries:

```
> db.addresses.find({ coordinates: { $near: [10, -87], $maxDistance : 1 }
})
{ "_id" : ObjectId("4f55abf8fed0eb2f6c00002e"), "_type" : "Address",
"coordinates" : [ 10.123123, -87.1231231 ], "location_id" : ObjectId("4
f55abf8fed0eb2f6c00002d"), "location_type" : "Author", "state" : "CA",
"street" : "101 Union Street", "zip" : nil }

> db.addresses.find({ coordinates: { $near: [10, -86], $maxDistance : 1 }
})

> db.addresses.find({ coordinates: { $near: [10, -88], $maxDistance : 1 }
})
{ "_id" : ObjectId("4f55abf8fed0eb2f6c00002e"), "_type" : "Address",
"coordinates" : [ 10.123123, -87.1231231 ], "location_id" : ObjectId("4
f55abf8fed0eb2f6c00002d"), "location_type" : "Author", "state" : "CA",
"street" : "101 Union Street", "zip" : nil }
```

We see that the address is not found within 1° of [10, -86]. Nice! Now let's keep the longitude the same and change the latitude by 1° in both directions:

```
> db.addresses.find({ coordinates: { $near: [11, -87], $maxDistance : 1 } })
{ "_id" : ObjectId("4f55abf8fed0eb2f6c00002e"), "_type" : "Address",
"coordinates" : [ 10.123123, -87.1231231 ], "location_id" : ObjectId("4
f55abf8fed0eb2f6c00002d"), "location_type" : "Author", "state" : "CA",
"street" : "101 Union Street", "zip" : nil }

> db.addresses.find({ coordinates: { $near: [9, -87], $maxDistance : 1 } })
{ "_id" : ObjectId("4f55abf8fed0eb2f6c00002e"), "_type" : "Address",
"coordinates" : [ 10.123123, -87.1231231 ], "location_id" : ObjectId("4
f55abf8fed0eb2f6c00002d"), "location_type" : "Author", "state" : "CA",
"street" : "101 Union Street", "zip" : nil }
```

Awesome! We see that for [9, -87], we don't get a result. The very fact that in some preceding cases, for a circular area of 1°, we are able to fetch the object and a fail implies that the `$near` query works now using `$maxDistance`.

Using mongoid_spacial

So how do we do this using Mongoid?



There is an interesting story to this. It was deemed better to keep geolocation queries for MongoDB in a separate gem to ensure that the `mongoid` gem remains "thin". So, the `mongoid_geo` gem was created. And if that was not enough, `mongoid_geo` has now evolved into `mongoid_spacial`.

Time for action – firing near queries in Mongoid

Let's add the gem to the Gemfile:

```
gem 'mongoid_spacial'
```

Now, for some minor changes in our code:

```
class Address
  include Mongoid::Document
  include Geocoder::Model::Mongoid
```

```

include Mongoid::Spatial::Document

field :street, type: String
...
field :coordinates, type: Array

spatial_index :coordinates
end

```

As we have already created indexes in the database, we don't need to run the `rake db:mongoid:create_indexes` command! Now, let's try our geolocation queries for the coordinates [10.123123, -87.123123]. Let's execute the following commands:

```

irb> Address.geo_near([10.923124, -87.8231232], max_distance: 1)
=> []

irb > Address.geo_near([10.923124, -87.8231232], max_distance: 2)
=> #<Address _id: 4f55abf8fed0eb2f6c00002e, _type: "Address", street:
"101 Union Street", zip: nil, city: "Pasedena", state: "CA", country:
"US", coordinates: [10.123123, -87.1231231], location_type: "Author",
location_id: BSON::ObjectId('4f55abf8fed0eb2f6c00002d')>

```

What just happened?

If we search within a distance equal to 1 radian around [10.92, -81.82], we don't find our address. But if we search within a distance of two radians, we find our address. So, it works! `mongoid_spatial` introduces a new criterion that taps the `$geoNear` operation in MongoDB.



`$geoNear` is available only from MongoDB v1.8 onwards

Let's take a few steps back and see what the difference is between `$near` and `$geoNear` in MongoDB.

Differences between `$near` and `$geoNear`

The earth is round but maps are flat.

In MongoDB, when we use 2D spatial indexing and use `$near`, it's like searching within a box or rectangle with the center of the box as the point we want to search with. Basically, the Pythagoras theorem is used to calculate the range of the box around the 2D point.

However, the earth is not flat but is a sphere. The longitudinal distances differ depending on the latitude. The default `$near` query does not cater to this as it is treated as a true 2D area for searching. So, the surface area changes when we consider a point on a sphere. This is what `$geoNear` does. It searches in a spherical manner and hence will give more accurate results when we use geospatial indexes.

Nothing would explain this better than an example:

```
irb> Address.geo_near([10.923124, -87.8231232], max_distance: 1)
=> []

irb> Address.geo_near([10.923124, -87.8231232], max_distance: 1,
spherical: true)
=> [#<Address _id: 4f55abf8fed0eb2f6c00002e, _type: "Address", street:
"101 Union Street", zip: nil, city: "Pasedena", state: "CA", country:
"US", coordinates: [10.123123, -87.1231231], location_type: "Author",
location_id: BSON::ObjectId('4f55abf8fed0eb2f6c00002d')>]
```

As we can see, just by adding an option `spherical`, MongoDB does a spherical search and the results change.

Summary

In this chapter, we have added geolocation to the `Address` model. We learned what is geolocation and how coordinates are mapped on the earth. We learned the use of `$near` and `$geoNear`, which do a box and a spherical search respectively. Finally, we plugged in the `geocoder` and `mongoid_spacial` gems for geolocation. You are now all set to build geolocation sensitive applications.

While you build your kick-ass application using MongoDB and Ruby, it's important to understand that scale should not hamper the growth of your web application. To be able to scale a web application and the database to millions of users, the right infrastructure is mandatory. MongoDB, as the name suggests, manages humongous data. Scalability is one of the powerful features that we shall learn in the next chapter.

10

Scaling MongoDB

This is the grand finale! Knowing how to use MongoDB is one thing but taking it to the next level—building large-scale applications, requires a lot more knowledge. In this chapter we shall see how we can use MongoDB to build large Internet applications.

In this chapter we will learn the following:

- ◆ Replication using master/slave configuration
- ◆ Replication using replica sets
- ◆ Scaling MongoDB using sharding
- ◆ High performance with large data using Map/Reduce

Scaling can be horizontal or vertical. **Vertical scaling** is when we upgrade the systems, by adding more memory, disk space, and CPUs. **Horizontal scaling** is when we add more commodity nodes or machines to the system. This chapter discusses how we can scale MongoDB horizontally!

By the end of this chapter we would have learned how to manage failover and high availability using MongoDB slaves and replica sets. We shall also see how we can use sharding to distribute the load across nodes when there are a huge number of documents. Finally, we shall see how we can use Map/Reduce techniques to collect and analyze large sets of data with high efficiency.

High availability and failover via replication

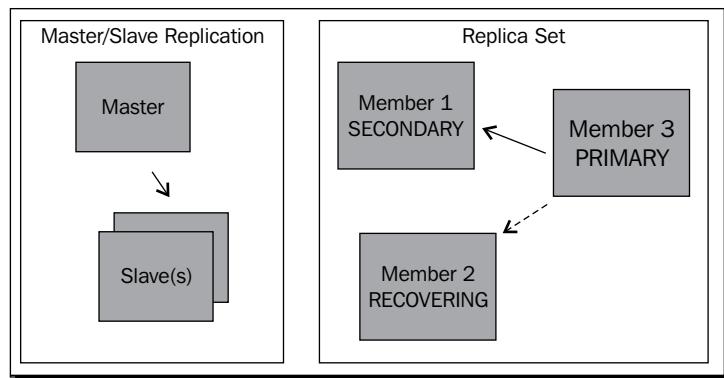
First let's understand what these terms mean.

High availability is when we can guarantee accessibility to the server. The higher the number of nodes that work together, the more the reliability and in turn, the availability of the system.

Failover is a term frequently used when a node in the system goes down and the request needs to be seamlessly handled by another node thereafter!

Replication, as the name suggests, is duplicating data on another node. This also adds redundancy to the system, that is there are more nodes with the same data and hence the chances of losing information due to machine failure is lesser.

There are two types of replication schemes in MongoDB—master/slave replication and replica sets, as shown in the following diagram:



Implementing the master/slave replication

This is standard practice with most databases. Typically there is one master and multiple slaves. This is also called the **active/passive** mode. All writes are only to the master and reads can be either from the master or slave. This ensures that there is write consistency with the database—which means that there will never be a case where data is written that will cause inconsistency in the database.

Time for action – setting up the master/slave replication

Let's set up the basic master/slave replication. We shall need two machines for this.

First, start the master:

```
server-1$ mongod --master
```

Now, we will start the slave server:

```
server-2$ mongod --slave --source server-1
```

That's it! Now we have `server-2` which is a slave of `server-1` and all the databases on `server-1` are seamlessly replicated to `server-2`.



In case `server-1` goes down, you need to change the configuration of the application to point to `server-2`.



What just happened?

We fired two simple commands and see that everything has started working. Let's understand them in detail:

```
$ sudo mongod --master -vvvv
```

This command will pick up the default `mongod.conf` file and start this server as the master!



Remember that `-vvvv` means very verbose. The more `v` you add, the more verbose output on the console.



If all is well, you should see this on the console:

```
[initandlisten] MongoDB starting : pid=53165 port=27017 dbpath=/usr/local/var/mongodb master=1 64-bit host=server-1
[initandlisten] db version v2.0.2, pdfile version 4.5
...
[initandlisten] Accessing: local for the first time
[initandlisten] query local.system.namespaces reslen:20 0ms
...
[initandlisten] master=true
[initandlisten] *****
[initandlisten] creating replication oplog of size: 183MB...
[initandlisten] create collection local.oplog.$main { size: 192000000.0, capped: true, autoIndexId: false }
[initandlisten] New namespace: local.oplog.$main
[initandlisten] New namespace: local.system.namespaces
...
[FileAllocator] allocating new datafile /usr/local/var/mongodb/local.ns, filling with zeroes...
[FileAllocator] creating directory /usr/local/var/mongodb/_tmp
[FileAllocator] done allocating datafile /usr/local/var/mongodb/local.ns, size: 16MB, took 2.174 secs
```

```
[FileAllocator] allocating new datafile /usr/local/var/mongodb/
local.0, filling with zeroes...
...
[initandlisten] runQuery called local.oplog.$main { query: {},
orderby: { $natural: -1 } }
[initandlisten] query local.oplog.$main ntoreturn:1 nscanned:1
nreturned:1 reslen:64 372ms
...
[initandlisten] waiting for connections on port 27017
[websvr] fd limit hard:9223372036854775807 soft:256 max conn: 204
[websvr] admin web console waiting for connections on port 28017
```

The console log we see is a very detailed one as it helps us understand how MongoDB replication works! Let's see this in smaller parts:

```
[initandlisten] master=true
[initandlisten] *****
[initandlisten] creating replication oplog of size: 183MB...
[initandlisten] create collection local.oplog.$main { size:
192000000.0, capped: true, autoIndexId: false }
[initandlisten] New namespace: local.oplog.$main
[initandlisten] New namespace: local.system.namespaces
```

We can see that the server has started as the master. The `local.oplog.$main` is a capped collection which saves all transaction log entries that will be replicated over to the slaves.

```
[FileAllocator] allocating new datafile /usr/local/var/mongodb/local.
ns, filling with zeroes...
[FileAllocator] creating directory /usr/local/var/mongodb/_tmp
[FileAllocator] done allocating datafile /usr/local/var/mongodb/local.
ns, size: 16MB, took 2.174 secs
```

When we set up the master for the first time, this `local.oplog.$main` capped collection and the local namespace is created (and depending on the machine this can take a few minutes!).

```
...
[initandlisten] runQuery called local.oplog.$main { query: {},
orderby: { $natural: -1 } }
[initandlisten] query local.oplog.$main ntoreturn:1 nscanned:1
nreturned:1 reslen:64 372ms
...
```

This is where the transaction logs are checked for their natural order and setup. After this, the master server is waiting for connections and serving requests normally.

Now let's see what happens when a slave connects:

```
$ sudo mongod --slave --source 192.168.1.141
[initandlisten] MongoDB starting : pid=20653 port=27017 dbpath=/usr/
local/var/mongodb slave=1 64-bit host=server-2
...
[replslave] repl: from host:192.168.1.141
[replslave] repl: applied 1 operations
[replslave] repl: end sync_pullOpLog syncedTo: Apr 5 15:33:41
4f7d6dfd:1
[replslave] repl: sleep 1 sec before next pass
```

At this point, the slave has sent a request to the master for syncing and received a reply. A lot of interesting things happen on the master:

```
[initandlisten] connection accepted from 192.168.1.153:63591 #1
[conn1] runQuery called admin.$cmd { handshake: ObjectId('4f7d6d3fb7d3
2a318178619f') }
[conn1] run command admin.$cmd { handshake: ObjectId('4f7d6d3fb7d32a3
18178619f') }
[conn1] command admin.$cmd command: { handshake: ObjectId('4f7d6d3fb7d
32a318178619f') } ntoreturn:1 reslen:37 0ms
[conn1] runQuery called local.oplog.$main { query: {}, orderby: {
$natural: -1 } }
[conn1] query local.oplog.$main ntoreturn:1 nreturned:1 reslen:64 0ms
```

This is the master/slave handshake and they exchange object IDs so that the master knows which slave has connected:

```
[conn1] runQuery called admin.$cmd { listDatabases: 1 }
[conn1] run command admin.$cmd { listDatabases: 1 }
[conn1] command: { listDatabases: 1 }
```

Next up, the master checks for which databases should be replicated:

```
[conn1] command admin.$cmd command: { listDatabases: 1 } ntoreturn:1
reslen:195 1143ms
[conn1] runQuery called local.oplog.$main { ts: { $gte: new
Date(5727855097040338945) } }
[conn1] query local.oplog.$main nreturned:1 reslen:64 47ms
BackgroundJob starting: SlaveTracking
```

Now, it checks the transaction log (local.oplog.\$main) to see where it should start the replication from and then spawns a SlaveTracking background job. This happens as follows:

```
[slaveTracking] New namespace: local.slaves
[slaveTracking] adding _id index for collection local.slaves
[slaveTracking] New namespace: local.system.indexes
```

```
[slaveTracking] build index local.slaves { _id: 1 }
mem info: before index start vsize: 3509 resident: 41 mapped: 544
[slaveTracking] external sort root: /usr/local/var/mongodb/_tmp/
esort.1333620219.2003184756/
mem info: before final sort vsize: 3509 resident: 41 mapped: 544
mem info: after final sort vsize: 3509 resident: 41 mapped: 544
[slaveTracking]      external sort used : 0 files  in 0 secs
[slaveTracking] New namespace: local.slaves.$_id_
[slaveTracking]      done building bottom layer, going to commit
[slaveTracking]      fastBuildIndex dupsToDrop:0
[slaveTracking] build index done 0 records 0.023 secs
```

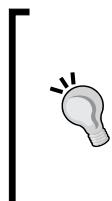
In case the local.slaves collection has not been built, the master builds it and indexes it:

```
[slaveTracking] update local.slaves query: { _id: ObjectId('4f7d6d3
fb7d32a318178619f'), host: "192.168.1.153", ns: "local.oplog.$main"
} update: { $set: { syncedTo: Timestamp 1333620189000|1 } }
fastmodinsert:1 134ms
```

Here, the slave is added with host information and its timestamp for replication. After this is done, there are continuous sync commands that would go back and forth between the master and the slave like this:

```
[conn1] getmore local.oplog.$main query: { ts: { $gte: new
Date(5727855097040338945) } } cursorid:1979419191886059940 reslen:20
2311ms
[conn1]    running multiple plans

[conn1] getmore local.oplog.$main query: { ts: { $gte: new
Date(5727855097040338945) } } cursorid:1979419191886059940 nreturned:1
reslen:64 886ms
```



The sync commands are continuous, they do not directly interfere with the routine database processing for the master, but they can consume valuable CPU and network resources.

It is recommended to keep the slave behind the master for an acceptable duration that depends on the application. We use the --slavedelay option for this.

What happens if the master goes down? The slave shows log entries like this:

```
[replslave] repl: from host:192.168.1.141
[replslave] repl: AssertionException dbclient error communicating with
server: 192.168.1.141
repl: sleep 2 sec before next pass
[replslave] repl: from host:192.168.1.141
```

```
[replslave] repl: couldn't connect to server 192.168.1.141
[replslave] repl: sleep 3 sec before next pass
[replslave] repl: from host:192.168.1.141
```

Once the master comes up again, the syncing begins.



It is possible to have a configuration such that the writes are always on the master but reads can be from the master or slave.



Suppose you want to simulate the master/slave configuration on a single machine, remember to run the slave on a different port

```
$ sudo mongod --slave --source localhost --port 27123
```

Using replica sets

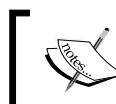
Using replica sets is the recommended approach for replication and failover.



Replica sets are available only in MongoDB versions after v1.6.

Replica sets, as the name suggests, are a bunch of MongoDB nodes that work together and keep replicas of the data. This is not a master/slave configuration! Nodes elect a leader, which then behaves as the master and the other nodes become the slaves and receive replication data. According to replica set terminology, they are called `PRIMARY` and `SECONDARY` respectively.

As this is the normal case for ensuring write consistency, we can write on to `PRIMARY` and if required read from `SECONDARY`. The beauty of replica sets is the election process. Nodes exchange handshakes and vote or veto nodes and finally elect a `PRIMARY`. We can also insert arbiters to ensure enough members for the voting process.



Arbiters are very light-weighted MongoDB instances that only vote! They are not replication nodes and are involved only in the voting process

Time for action – implementing replica sets

We can simulate replica sets on a single machine too. We need three different terminals for this—Terminal 1, Terminal 2, and Terminal 3, to start the three different MongoDB processes:

```
Term-1 $ sudo mongod --replSet sodibee --port 27017 --dbpath /data/repl1
```

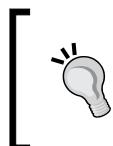
```
Term-2 $ sudo mongod --replSet sodibee --port 27018 --dbpath /data/repl2
```

```
Term-3 $ sudo mongod --replSet sodibee --port 27019 --dbpath /data/repl3
```

Notice, that the replica set has the same name in all instances. As we are running this on the same machine, we need to specify different ports. The default port is fine if running on different instances. Once these are started, on the database console, we shall see something like this:

```
[initandlisten] MongoDB starting : pid=21876 port=27017 dbpath=/data/
repl1 64-bit host=gautam-2.local
[initandlisten] db version v2.0.2, pdfile version 4.5
...
[rsStart] sodibee can't get local.system.replset config from self or
any seed (EMPTYCONFIG)
[rsStart] sodibee info you may need to run replSetInitiate --
rs.initiate() in the shell -- if that is not already done
```

As we can see, just starting them up (like in the case of the master/slave configuration) is not enough! We need to initialize the replica sets. To do this, we need to login to the PRIMARY, that is, the node we want to replicate to the other MongoDB instances.



Remember, that the MongoDB instance you initiate the replication command will be PRIMARY at first. The SECONDARY nodes have to have a clean dbpath, that is, they cannot have existing data! All members of the replica sets must be empty except the initiator!

Let's execute the following commands:

```
$ mongo localhost:27017
MongoDB shell version: 2.0.2
connecting to: localhost:27017/test

> config = {_id: 'sodibee', members: [
    {_id: 0, host: 'localhost:27017'},
    {_id: 1, host: 'localhost:27018'},
    {_id: 2, host: 'localhost:27019'}
```

```
    ],
  {
    "_id" : "sodibee",
    "members" : [
      {
        "_id" : 0,
        "host" : "localhost:27017"
      },
      {
        "_id" : 1,
        "host" : "localhost:27018"
      },
      {
        "_id" : 2,
        "host" : "localhost:27019"
      }
    ]
  }

> rs.initiate(config);
{
  "info" : "Config now saved locally. Should come online in about a
minute.",
  "ok" : 1
}
```

This instantiates the replica sets and we are all set!

What just happened?

We started three instances of MongoDB with the `--replSet` option. Then we initialized the replica sets and we were on our way. Let's see what happened here!

```
> config = {_id: sodibee, members: [
    {_id: 0, host: 'localhost:27017'},
    {_id: 1, host: 'localhost:27018'},
    {_id: 2, host: 'localhost:27019'}
]}
```

This is the configuration we have explicitly set up, as per the MongoDB instances we configured earlier. Then we need to initialize them:

```
> rs.initiate(config);
{
    "info" : "Config now saved locally. Should come online in about a
minute.",
    "ok" : 1
}
```

When we run this `initiate` command with the configuration we have specified, voting between the replica sets takes place and they elect a primary. The following is what we see on the MongoDB console which we connected to initiate replica sets:

```
[conn2] sodibee replSetInitiate admin command received from client
[conn2] sodibee replSetInitiate config object parses ok, 3 members
specified
[conn2] sodibee replSetInitiate all members seem up
[conn2] *****
[conn2] creating replication oplog of size: 183MB...
[FileAllocator] allocating new datafile /data/repl1/local.ns, filling
with zeroes...
```

This is what we see on the other MongoDB consoles:

```
[rsStart] trying to contact localhost:27017
[rsStart] sodibee got config version 1 from a remote, saving locally
[rsStart] sodibee info saving a newer config version to local.system.
replset
[FileAllocator] allocating new datafile /data/repl2/local.ns, filling
with zeroes...
```

Basically, every instance is setting up their local systems for saving information. After this the voting process begins. We see messages like the following, on the node that becomes the PRIMARY node:

```
[rsMgr] replSet PRIMARY
[rsSync] replSet SECONDARY
[rsMgr] not electing self, localhost:27019 would veto
[rsMgr] replSet info electSelf 0
[rsMgr] replSet PRIMARY
```

And, we see messages like this on the nodes which become SECONDARY:

```
[rsStart] sodibee saveConfigLocally done
[rsStart] replSet STARTUP2
[rsSync] *****
[rsSync] creating replication oplog of size: 183MB...
```

```
[rsHealthPoll] replSet member localhost:27017 is up
[rsHealthPoll] replSet member localhost:27017 is now in state
SECONDARY
[rsHealthPoll] replSet member localhost:27019 is up
[rsHealthPoll] replSet member localhost:27019 is now in state STARTUP2
[conn4] sodibee info voting yea for localhost:27017 (0)
[rsHealthPoll] replSet member localhost:27019 is now in state
RECOVERING
[conn4] sodibee info voting yea for localhost:27017 (0)
[rsHealthPoll] replSet member localhost:27017 is now in state PRIMARY
```

To see if a MongoDB node is primary or secondary, we can connect to any MongoDB node and execute the following command:

```
$ mongo localhost:27019
MongoDB shell version: 2.0.2
connecting to: localhost:27019/test

SECONDARY> rs.status()
```

As we can see, when we connect to a node, it tells us if the node was a PRIMARY or a SECONDARY. In the preceding case, we connected to a secondary. `rs.status()` tells us the status of the replica sets. The result of the `rs.status()` command is given as follows:

```
{
  "set" : "replSet",
  "date" : ISODate("2012-04-06T07:18:56Z"),
  "myState" : 2,
  "syncingTo" : "localhost:27017",
  "members" : [
    {
      "_id" : 0,
      "name" : "localhost:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 139,
      "optime" : {
        "t" : 1333696634000,
        "i" : 1
      },
      "optimeDate" : ISODate("2012-04-06T07:17:14Z"),
      "lastHeartbeat" : ISODate("2012-04-06T07:18:55Z"),
      "pingMs" : 0
    },
    ...
  ]
}
```

```
{  
    "_id" : 1,  
    "name" : "localhost:27018",  
    "health" : 1,  
    "state" : 2,  
    "stateStr" : "SECONDARY",  
    "uptime" : 141,  
    "optime" : {  
        "t" : 1333696634000,  
        "i" : 1  
    },  
    "optimeDate" : ISODate("2012-04-06T07:17:14Z"),  
    "lastHeartbeat" : ISODate("2012-04-06T07:18:55Z"),  
    "pingMs" : 0  
},  
{  
    "_id" : 2,  
    "name" : "localhost:27019",  
    "health" : 1,  
    "state" : 2,  
    "stateStr" : "SECONDARY",  
    "optime" : {  
        "t" : 1333696634000,  
        "i" : 1  
    },  
    "optimeDate" : ISODate("2012-04-06T07:17:14Z"),  
    "self" : true  
}  
],  
"ok" : 1  
}
```

As we can see, there is always only one PRIMARY and the other nodes will sync with this PRIMARY. Let's now see how we access and write data! Execute the following commands:

```
$ mongo localhost:27017  
MongoDB shell version: 2.0.2  
connecting to: localhost:27017/test  
  
PRIMARY> db.messages.insert({name: "Sodibee works!"});  
PRIMARY>  
PRIMARY> db.messages.find()  
{ "_id" : ObjectId("4f7e921f9f044ed2db843466") , "name" : "Sodibee works!" }
```

Let's see what happens if we try to read and write from a SECONDARY:

```
$ mongo localhost:27018
MongoDB shell version: 2.0.2
connecting to: localhost:27018/test

SECONDARY> db.messages.find()
error: { "$err" : "not master and slaveok=false", "code" : 13435 }
```

As this is the secondary, we cannot read or write to it, it's just for replication! But if we really do want to read from the SECONDARY to improve read performance, we can configure it using `rs.slaveOk()`, shown as follows:

```
SECONDARY> rs.slaveOk();
not master and slaveok=false

SECONDARY> db.messages.find()
{ "_id" : ObjectId("4f7e921f9f044ed2db843466") , "name" : "Sodibee works!" }
```

Recovering from crashes – failover

What happens if the PRIMARY crashes or shuts down? We can easily simulate this by either killing the PRIMARY or if it's running in foreground, press `Ctrl + C`. The replica sets detect that the PRIMARY is down and vote among each other to become the PRIMARY node! We can see something like this on the console:

```
[rsHealthPoll] sodibee member localhost:27017 is now in state DOWN
[rsMgr] not electing self, localhost:27019 would veto
[conn21] sodibee info voting yea for localhost:27019 (2)
[rsHealthPoll] sodibee member localhost:27019 is now in state PRIMARY
```

As we can see the PRIMARY changed automatically.

Adding members to the replica set

Now, suppose we have started up with three members in a replica set and we need to scale up with one more, it's very easy to do so! First start a new MongoDB instance on a different machine or on the same machine on a different port. This is done as follows:

```
$ sudo mongod --replSet sodibee --port 27020 --dbpath /data/repl4
```

We need to add this to the replica set configuration. So, we connect to the PRIMARY and reconfigure the replica set. This is done by executing the following commands:

```
$ mongo  
MongoDB shell version: 2.0.2  
connecting to: test  
  
PRIMARY> rs.add("localhost:27020")  
{ "ok" : 1 }
```

Voila! You just scaled up the setup. This will automatically start the replication process as a SECONDARY for the new node.

Implementing replica sets for Sodibee

So far so good! How do we use these replica sets in our Ruby web application? Let's see how we can use replica sets in Sodibee!

Time for action – configuring replica sets for Sodibee

Let's restart MongoDB service as a replica set:

```
$ sudo mongod --rest -vvvv --replSet sodibee
```

Note, that the command is the same as it was for the master/slave except for the additional --replSet option! Now also start the other MongoDB instance to be part of the replica set. In our case, let's simulate this on a single host. So, we shall start this MongoDB instance on a different port:

```
$ sudo mongod --replSet sodibee --port 27019 --dbpath /data/sodibee1
```

Now these two instances are set up, all we need to do is initiate the replica sets and get started! Let's do that!



[It's strongly recommended to have at least three members in a replica set. As we shall soon see, this is needed to ensure a quorum during the voting process!]

Let's execute the following commands:

```
$ mongo  
MongoDB shell version: 2.0.2  
connecting to: test
```

```
> config = { _id: 'sodibee', members: [
... { _id: 0, host: 'localhost:27017' },
... { _id: 1, host: 'localhost:27019' }
... ]}
{
  "_id" : "sodibee",
  "members" : [
    {
      "_id" : 0,
      "host" : "localhost:27017"
    },
    {
      "_id" : 1,
      "host" : "localhost:27019"
    }
  ]
}

> rs.initiate(config)
{
  "info" : "Config now saved locally. Should come online in about a
minute.",
  "ok" : 1
}
```

Now these two MongoDB replica sets will "talk" to each other and become the PRIMARY and SECONDARY automatically.

Let's configure config/mongoid.yml now with this new configuration. This is done as follows:

```
development:
  database: sodibee_development
  hosts:
    - - localhost
    - 27017
    - - localhost
    - 27019
  read_secondary: true
```

That's it! Restart the server and we are done! Let's test this out. Let's say we are editing the details of an author, as shown in the following screenshot:

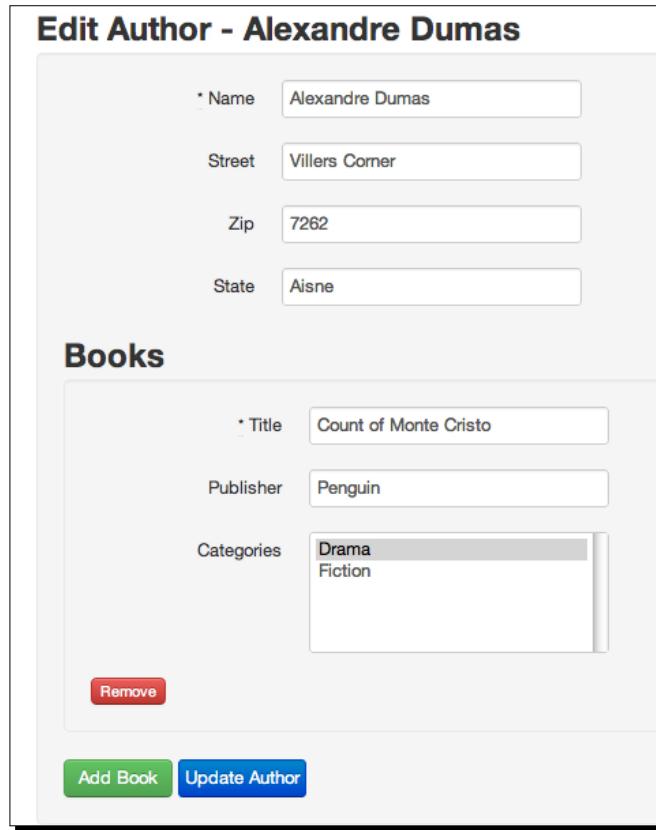
Edit Author - Alexandre Dumas

* Name	Alexandre Dumas
Street	Villers Corner
Zip	7262
State	Aisne

Books

* Title	Count of Monte Cristo
Publisher	Penguin
Categories	Drama Fiction

Buttons: Remove, Add Book, Update Author



While doing so, before we can click on the **Update Author** button, the PRIMARY crashes! (In our case, we do a *Ctrl + C* and stop it). Now two things can happen:

- ◆ We refresh the page before the SECONDARY becomes PRIMARY (in those few seconds of a changeover)
- ◆ We wait for a few seconds after which the SECONDARY becomes PRIMARY

In case we don't wait long enough, we could see an exception, as shown in the following screenshot:



This exception is because the current connection is not resolved! Refresh the page and it should start working! If we do that however, much to our chagrin, we see another exception, as shown in the following screenshot:



Jeez! This is not working. Let's do something different. Let's add a third member to this replica set:

```
$ sudo mongod --replSet sodibee --port 27018 --dbpath /data/sodibee2
```

Let's also add this to our replica sets:

```
$ mongo
MongoDB shell version: 2.0.2
connecting to: test

PRIMARY> rs.add("localhost:27018")
{ "ok" : 1 }

PRIMARY>
```

Now, reconfigure `mongoid.yml` to add this third member:

```
development:  
  database: sodibee_development  
  hosts:  
    - - localhost  
      - 27017  
    - - localhost  
      - 27018  
    - - localhost  
      - 27019  
  read_secondary: true
```

Restart the server and refresh the page. It works now!

What just happened?

When a MongoDB connection is lost, Mongoid automatically creates another connection with the next PRIMARY node in the replica set. This can take a few seconds during which we get some connection-reset errors. Considering a web application, this is fine!

When working with MongoDB replica sets, never work only with two nodes! It's always advisable to work with at least three members in our replica set. These are three MongoDB instances or three members with one member being an arbiter!

This is important because in a voting scenario, we need a majority to make a node a PRIMARY! If we have only two members in a replica set and one of them goes down, we don't have a majority to promote the other node as the PRIMARY. In such a case you would see a console log like this:

```
[rsMgr] can't see a majority of the set, relinquishing primary  
[rsMgr] replSet relinquishing primary state  
[rsMgr] replSet SECONDARY  
[rsMgr] replSet closing client sockets after relinquishing primary
```

In our earlier case when we had only two members, we saw the couldn't connect to server (that is, the primary node) exception, precisely for this reason. When we added a third member to the set, one of them became a PRIMARY and things started working.

We could have started the third instance only as an arbiter if we don't really want to replicate data more than twice.

```
PRIMARY> rs.add("localhost:27018", arbiterOnly: true)
```

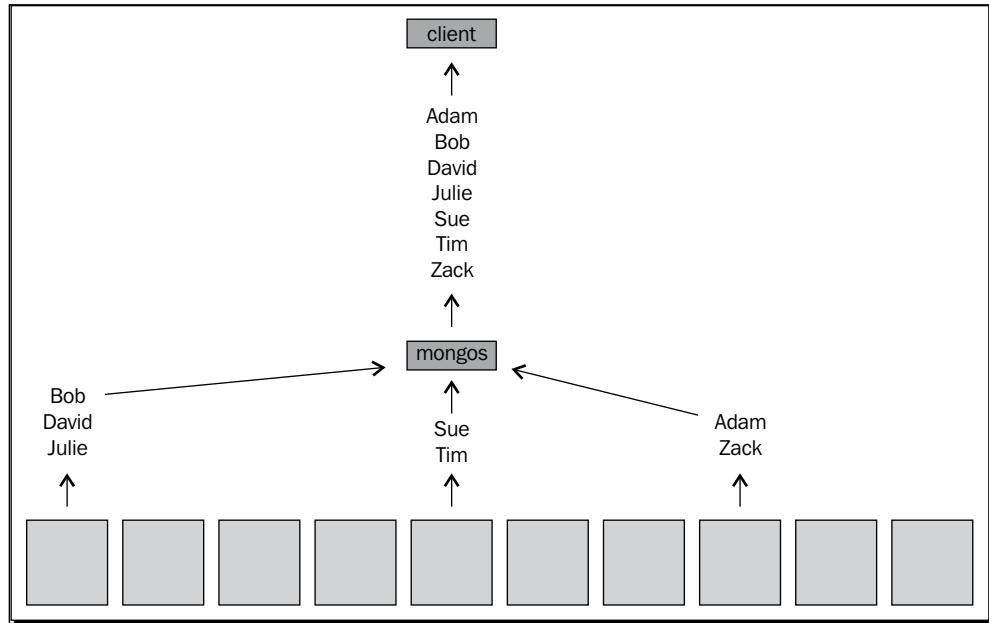
Implementing sharding

Sharding is the real horizontal scaling out. Replication is to ensure data safety, failover, and high availability. Both are configured in a similar way and work in conjunction, but are conceptually very different!

Sharding is where we distribute the data among various MongoDB instances, not replicate but distribute! So, in Sodibee, we can distribute the authors based on their names.

In real-world scenarios, tweets of different people can be sharded and stored in different servers. Twitter uses MySQL sharding using Gizzard.
Read more here (<http://engineering.twitter.com/2010/04/introducing-gizzard-framework-for.html>)
PostgreSQL provides partitioning which is the same as sharding in MongoDB. Read more about it at <http://www.postgresql.org/docs/current/interactive/ddl-partitioning.html>.

To give you an idea of how sharding would take place, take a look at the following diagram:



Basically, all names of authors would be stored in different MongoDB instances based on some criteria, called a **shard key**. In the preceding diagram the shard key is the name! The client does not even realize that the results are coming from a shard. This greatly improves the performance of reads and writes!

Creating the shards

As we have seen, sharding and replication are different. One of the ways to get the best of replication and sharding is combining them and using a sharded replica set! Let's see how this is done!

Time for action – setting up the shards

Let's see how we can set up shards. Ideally, we should use different machines, but we can do that on a single machine for now!

First, we need to start the MongoDB instances with the `--shardsvr` option:

```
$ sudo mongod --shardsvr --port 27025 --dbpath /data/shard2
```

This is one of our new shard servers running on port 27025. As we already have a replica set created earlier, we shall create a replicated shard with it! Just like earlier, we add the `--shardsvr` option to it too:

```
$ sudo mongod --replSet sodibee --port 27018 --dbpath /data/sodibee2  
--shardsvr
```

Let's have three replica sets configured with this shard running on ports 27018, 29019, and 27020. This is done as follows:

```
$ mongo localhost:27018  
MongoDB shell version: 2.0.2  
connecting to: localhost:27018/test  
  
PRIMARY> rs.config()  
{  
  "_id" : "sodibee",  
  "version" : 4,  
  "members" : [  
    {  
      "_id" : 1,  
      "host" : "localhost:27019"  
    },  
    {  
      "_id" : 2,  
      "host" : "localhost:27018"  
    },  
    {  
      "_id" : 3,  
      "host" : "localhost:27020"  
    }  
  ]  
}  
PRIMARY>
```

```
{
  "_id" : 3,
  "host" : "localhost:27020"
}
]
```

What just happened?

We now have two shards:

- ◆ One is a standalone MongoDB instance running on port 27025
- ◆ One is a sharded replica set with the name sodibee

Configuring the shards with a config server

The config server is the central server that has information about where all the shards reside. All nodes communicate with the config server to know who is in the system.

Time for action – starting the config server

Start another MongoDB instance with the --configsvr flag:

```
$ sudo mongod -vvvv --configsvr --port 27200
```

The default port is 27019, so we specify a different port 27200, as 27019 is already used by one of the shards. We now need to set up the sharding configuration on this server. This is done as follows:

```
$ mongo
MongoDB shell version: 2.0.2
connecting to: test

mongos> use admin
switched to db admin

mongos> db.runCommand( { addshard: "localhost:27025" } )
{ "shardAdded" : "shard0000", "ok" : 1 }

mongos> db.runCommand( { addshard: "sodibee/localhost:27018,localhost:27019,localhost:27020" } )
{ "shardAdded" : "sodibee", "ok" : 1 }
```



Notice the difference in syntax while adding a shard and a shard with replica sets!

Now, we need to enable sharding for the database:

```
mongos> db.runCommand( { enablesharding: "sodibee_development" } )  
{ "ok" : 1 }
```

Finally, we need to configure the shard key. In our case, we shall configure it for the author names! We can do this as follows:

```
mongos> db.runCommand( { shardcollection : "sodibee_development.authors",  
key : {name : 1  
{ "collectionsharded" : "sodibee_development.authors", "ok" : 1 }
```

What just happened?

We are almost set now. We have started the configuration server and loaded the options for sharding. We are sharding on the author name here. It's important to remember some rules here:

- ◆ The shard key should be unique so as to ensure consistency
- ◆ Shard keys are immutable, that is, they cannot be changed
- ◆ Never query a shard directly, as it will return only partial results. Each shard is, after all, a MongoDB instance
- ◆ Prior to v2.0 sharding was not secure. Post v2.0 sharding has an authentication mode

Setting up the routing service – mongos

The `mongos` process is the routing service for a MongoDB cluster. This basically "talks" to the config server. It is not a MongoDB instance but a non-persistent router. It gets all its information from the config server. It also acts as the load balancer.

Time for action – setting up mongos

For all servers that need to connect to this MongoDB cluster, it should go via this `mongos` router! First start it up with the configuration server details:

```
$ sudo mongos --configdb localhost:27200 --chunkSize 1
```

Now, this service will listen on the default 27017 port.

What just happened?

After you start `mongos`, you should see something like this on the console:

```
mongos db version v2.0.2, pdfile version 4.5 starting (--help for
usage)
...
[Balancer] about to contact config servers and shards
[mongosMain] waiting for connections on port 27017
[Balancer] updated set (sodibee) to: sodibee/
localhost:27018,localhost:27020
[Balancer] updated set (sodibee) to: sodibee/localhost:27018,localhost
:27020,localhost:27019
[ReplicaSetMonitorWatcher] starting
[Balancer] config servers and shards contacted successfully
...

```

Notice that `mongos` now waits for client connections and has contacted the config servers and shards. It now knows where to send the incoming requests for getting results.



The default chunk size is 64 MB. In order to simulate sharding I have kept it at 1 MB using the option `--chunkSize`.



Now, all that remains is to configure our Rails server to talk to `mongos` instead of the replica sets directly. Basically reset the configuration back to:

```
development:
  host: localhost
  database: sodibee_development
```



Configuring Mongoid models for the shard key

We have configured, in our example, the sharding on the `authors` collection and the shard key is the author's name. This should be reflected in the models.

The shard key should be indexed.



Make the relevant change in the models to reflect the shard key:

```
class Author
  include Mongoid::Document

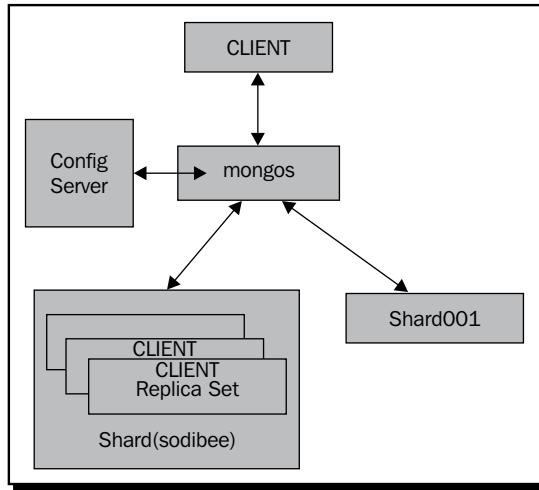
  ...

  index :name
  shard_key :name
end
```

Start your engines, that is, restart the Rails server and the data will be automatically sharded and replicated.

Testing shared replication

The process we just saw is depicted in the following diagram:



When a request is sent to the `mongos` server, it looks up the config server and reads information about the shards. Then, depending on the request and shard key, it sends the request to the relevant shard.

How do we see what is getting sharded? Or how do we know it's really getting sharded? Well, you won't from the web application. But you can execute some administrative commands and find out:

```
$ mongo  
MongoDB shell version: 2.0.2  
connecting to: test  
  
mongos> db.printShardingStatus()
```

You should see something like the following:

```
--- Sharding Status ---  
sharding version: { "_id" : 1, "version" : 3 }  
shards:  
{ "_id" : "shard0000", "host" : "localhost:27025" }  
{ "_id" : "sodibee", "host" : "sodibee/localhost:27018,localhost:27020,localhost:27019" }
```

```
databases:
  {
    "_id" : "admin", "partitioned" : false, "primary" : "config"
  }
  {
    "_id" : "sodibee_development", "partitioned" : true,
    "primary" : "sodibee" }
      sodibee_development.authors chunks:
        sodibee      2
        shard0000     1
        { "name" : { $minKey : 1 } } --> {
          "name" :
          "000094f21fd7d6af713da9e5ba1fc23b30f283d4632a12f3a88ff4518dcdfa30"
        } on : sodibee { "t" : 2000, "i" : 1 }
        {
          "name" :
          "000094f21fd7d6af713da9e5ba1fc23b30f283d4632a12f3a88ff4518dcdfa30"
        } --> {
          "name" :
          "fffff7bbf1dc325ce05d5be442d24ee26a1ab33ffb9663cfb4449a8c7d564a888"
        } on : sodibee { "t" : 1000, "i" : 3 }
        {
          "name" :
          "fffff7bbf1dc325ce05d5be442d24ee26a1ab33ffb9663cfb4449a8c7d564a888"
        } --> { "name" : { $maxKey : 1 } } on : shard0000 { "t" : 2000, "i" :
0 }
        { "_id" : "sodibee", "partitioned" : true, "primary" :
"shard0000" }
```

Notice that the collection data is sharded between two nodes, out of which one is a replica set!

Implementing Map/Reduce

Until now, we have seen how to ensure that our data is safe using replica sets. We have also seen how to shard data so that the distributed system can scale! Along with scale, we also want to ensure that we do not degrade our performance over a large set of data. This is where Map/Reduce comes into the picture. We have discussed what Map/Reduce is earlier in the book. Now, we see it practically and see how it makes sense to be used!

We have seen earlier the concept of Map/Reduce. Let's refresh it briefly. We can "map" our data into multiple independent tasks, process the temporary results and "reduce" the results in parallel. Basically, we spawn many parallel tasks to mappers. These mappers (which can be threads, processes, servers, among others) process a specific dataset and spew out results to the reducers. As the reducers keep getting information, they update the final results with this data.

This is how massively parallel processing is done! In MongoDB, `map` and `reduce` functions are written as JavaScript functions. Using the evented nature of JavaScript, Map/Reduce is a very handy ingrained functionality of MongoDB.

Time for action – planning the Map/Reduce functionality

In Sodibee, suppose we want to show the statistical count of authors by the starting alphabet of their name, it is a good case for using Map/Reduce. We want to see information like this:

```
Authors starting with "a": 1020
Authors starting with "b": 477
Authors starting with "c": 719
Authors starting with "d": 586
Authors starting with "e": 678
```

First, let's create many authors in our database. For this we shall use the `faker` gem, so that we can generate nice names. This is the `rake` task that we can use to generate ten thousand authors:

```
require 'faker'
task :fake_authors => :environment do
  10000.times do
    a = Author.create(:name => "#{Faker::Name.first_name}
#{Faker::Name.last_name}")
  end
end
```

To run this, we simply use `rake`:

```
$ bundle exec rake fake_authors
```

What just happened?

This should have created 10,000 authors in our database. Test and check that authors are getting created correctly from the rails console:

```
$ rails c
irb > Author.limit(5).collect(&:name)
=> ["Victor Metz", "Dayana Rau", "Ada Wiza", "Price Osinski", "Virgie
Hand"]
```

First, let's see how this could work in the MongoDB console. In our case, the `map` function is to get the name of the author. They emit the result for the first letter of the author's name. For example, if the authors name is "Charles Dickens", we want to emit the key as "c" and the count as 1.

Time for action – Map/Reduce via the mongo console

Let's execute the following commands:

```
$ mongo
MongoDB shell version: 2.0.2
connecting to: test

mongos> use sodibee_development
switched to db sodibee_development

mongos> map = function () {
    emit(this.name.toLowerCase()[0], {count:1});
}

mongos> reduce = function (key, values) {
    var r = {count:0};
    values.forEach(function (value) {
        r.count += value.count;
    });
    return r;
}

mongos> res = db.authors.mapReduce(map, reduce, { out: "authors_dr" } );

mongos> db.authors_dr.find()
{ "_id" : "a", "value" : { "count" : 1020 } }
{ "_id" : "b", "value" : { "count" : 477 } }
{ "_id" : "c", "value" : { "count" : 719 } }
{ "_id" : "d", "value" : { "count" : 586 } }
{ "_id" : "e", "value" : { "count" : 678 } }
{ "_id" : "f", "value" : { "count" : 240 } }
{ "_id" : "g", "value" : { "count" : 396 } }
...
```

What just happened?

Running a Map/Reduce task is about the `map` function and the reducer. Let's see this in detail:

```
map = function () {
    emit(this.name.toLowerCase()[0], {count:1});
}
```

This function will be executed for each `Author` document. It first takes the name and converts it to lowercase. Then, it emits the first character of the name along with the count as 1.

The `reduce` function looks like the following:

```
reduce = function (key, values) {
    var r = {count:0};
    values.forEach(function (value) {
        r.count += value.count;
    });
    return r;
}
```

The `reduce` function takes two parameters: the `key` that was emitted and an array of the `values` for this particular key.

A `map` function is executed once for each member of the dataset. In case of reducers however, it is given an array of results emitted by the mapper function as well as the temporary reduced results.

For example, suppose we have 10 authors starting with "a". There would be 10 results emitted by the mappers. However, when the reducer function is called, it would be given the emitted result that is `{ count: 1 }` along with a temporary reduced result, `{ count: 8 }`.



It's very important not to assume that the value passed to the reducers is the same as that emitted from the map function. In most cases, it would be different.

This is what the result of the `mapReduce` function looks like:

```
mongos> res = db.authors.mapReduce(map, reduce, { out: "authors_dr" })
;
{
    "result" : "authors_dr",
    "shardCounts" : {
        "localhost:27025" : {
            "input" : 0,
            "emit" : 0,
            "reduce" : 0,
```

```

        "output" : 0
    },
"sodibee/localhost:27018,localhost:27020,localhost:27019" : {
    "input" : 10000,
    "emit" : 10000,
    "reduce" : 251,
    "output" : 26
}
},
"counts" : {
    "emit" : NumberLong(10000),
    "input" : NumberLong(10000),
    "output" : NumberLong(26),
    "reduce" : NumberLong(251)
},
"ok" : 1,
"timeMillis" : 980,
"timing" : {
    "shards" : 633,
    "final" : 346
},
}
}

```

As we can see, there are 10,000 emitted results but only 251 reducer invocations!



In a sharded environment, MongoDB automatically distributes the `map` functions if the input collection is sharded. By default, the output collection of the `reduce` function is not shared and remains on one of the shards.

It's interesting to note that the request for 10,000 nodes went to only one shard because the data is stored on that node only. If the chunk size increases beyond that value set in the configuration, then it will get sharded.

Implementing this in Ruby is no different from MongoDB. As we have to pass the JavaScript functions to MongoDB, we do it via strings!

Time for action – Map/Reduce via Ruby

We modify the `Author` model to help us generate statistical data, as follows:

```

class Author
  include Mongoid::Document
  ...

```

```
def self.statistics
  map = %q{function() {
    emit(this.name.toLowerCase()[0], {count:1});
  }
}

reduce = %q{function(key, values) {
  var r = { count: 0 };
  values.forEach(function(value) {
    r.count += value.count;
  })
  return r;
}
}

res = Author.collection.map_reduce(map, reduce, out: "author_stats")
end
end
```

As we can see, the functions are exactly the same as those that we tried out on the MongoDB console! Let's run this:

```
$ rails c
Loading development environment (Rails 3.2.0)

irb> res = Author.statistics
=> #<Mongo::Collection:0x1cd25ac @name="author_stats", @
db=#<Mongo::DB:0x1fef8ac @name="sodibee_development",
...
> res.find().to_a
=> [
{"_id"=>"a", "value"=>{"count"=>1028.0}},
 {"_id"=>"b", "value"=>{"count"=>352.0}},
 {"_id"=>"c", "value"=>{"count"=>1164.0}},
 {"_id"=>"d", "value"=>{"count"=>932.0}},
 {"_id"=>"e", "value"=>{"count"=>162.0}},
 {"_id"=>"f", "value"=>{"count"=>1336.0}},
 {"_id"=>"g", "value"=>{"count"=>1393.0}},
...
]
```

What just happened?

This gives us the output we need. How do we know that all the authors were indeed computed? Let's execute the following command to find out:

```
> res.find().to_a.inject(0) do |sum, e|
...     sum + e["value"]["count"]
... end
=> 10000.0
```

Performance benchmarking

You may ask, is it really worth the effort to do a `mapReduce`? Why not just access all the objects and iterate? How much difference would it actually make? A world of difference!

Time for action – iterating Ruby objects

If we had to write this function in plain Ruby using iterations, we would write something like this:

```
class Author
  include Mongoid::Document
  ...

  def self.statistics_depr
    matches = {}
    Author.all.each do | a |
      key = a.name.downcase.first
      matches[key] = matches[key].to_i + 1
    end
    matches
  end
end
```



Ruby has a module called "Benchmark" which helps us find out the real time for any method call.



Let's benchmark Ruby object processing and `mapReduce` calls:

```
$ rails c

irb> Author.count
=> 10000
```

```
irb> Benchmark.realtime { Author.statistics }
=> 1.116757869720459

irb> Benchmark.realtime { Author.statistics_depr }
=> 1.9303243160247803
```

Let's increase the number of authors to 30,000 now by invoking `rake` twice:

```
$ rake fake_authors
$ rake fake_authors
```

Now, let's see the benchmarks:

```
irb> Author.count
=> 30000

irb> Benchmark.realtime { Author.statistics }
=> 1.4425742626190186

irb> Benchmark.realtime { Author.statistics_depr }
=> 6.486238956451416
```

What just happened?

We just saw the power of Map/Reduce. It took approximately 6.5 seconds to iterate the Ruby objects where as it took 1.44 seconds to run the `mapReduce` function. If we see this in more detail, as the scale increases, see how skewed the results are:

Number of authors	Map/Reduce	Ruby iteration
10,000	1.116 seconds	1.930 seconds
30,000	1.442 seconds	6.486 seconds
50,000	2.087 seconds	10.422 seconds
70,000	2.921 seconds	14.228 seconds
100,000	4.017 seconds	21.217 seconds

Needless to say, Map/Reduce is indeed very helpful.

Pop quiz – scaling our web app

1. How does MongoDB scale as a database?
 - a. Vertically and Horizontally.
 - b. Horizontally.
 - c. Vertically.
 - d. Diagonally.
2. Which of the following is incorrect for a master/slave configuration?
 - a. There must be only one master and many slaves.
 - b. Slaves are always read-only that is, we cannot write to them.
 - c. Slaves will elect a master automatically if the master crashes.
 - d. Slaves can be added anytime to the setup.
3. Which of the following is true for replica sets?
 - a. You must have at least three nodes for replica sets to start with replication process.
 - b. When the PRIMARY fails, you should have at least three nodes in the replica set to elect a new PRIMARY.
 - c. For the voting process, you must have at least one arbiter node in a replica set.
 - d. When the failed PRIMARY comes up again, it regains ownership as the PRIMARY.
4. What effect does the --chunkSize option in sharding have?
 - a. It sets the size of the chunk in MB, so that the documents are distributed when that threshold is crossed.
 - b. Chunk size is the amount of data fetched from the shard.
 - c. Chunk size determines the number of shards in the setup.
 - d. Chunk size is the maximum size of the document chunk that is stored in each shard.
5. Why does the reduce function take the key and a values array as a parameter?
 - a. One key will have many different values.
 - b. values array contains temporary results as well as emitted results for that key.
 - c. The map function emits an array, so the reduce function processes an array.
 - d. All the emitted values are passed to the reduce function in the array.

Summary

In this chapter, we have seen various important aspects about data—safety, scaling, and performance under scaling. We have seen how we can replicate data using a master/slave configuration. We can create replica sets for failover and high availability and how we can scale using shards and even shared replica sets! We saw how efficient Map/Reduce functions are with large datasets.

This does indeed bring us to the very end of the journey. I hope this book can help you build large scale web applications using Ruby and MongoDB.

Pop Quiz Answers

Chapter 2: Diving Deep into MongoDB

1	2	3	4	5	6
b	a	b	c	c	a

Chapter 3: MongoDB Internals

1	2	3	4
b	a	c	b

Chapter 4: Working out your Way with Queries

1	2	3	4
b	a	d	b

Chapter 5: Ruby DataMappers: Ruby and MongoDB Go Hand in Hand

1	2	3	4
d	b	c	a

Chapter 6: Modeling Ruby with Mongoid

1	2	3	4	5
d	c	a	d	b

Chapter 8: Rack, Sinatra, Rails and MongoDB - Making use of them all

1	2	3	4	5
a	b	c	d	d

Chapter 10: Scaling MongoDB

1	2	3	4	5
b	c	b	a	b

Index

Symbols

\$exists
used, for checking presence 89
\$geoNear query 264
\$gt 89
\$gte 89
\$in and \$nin
used, for searching inside arrays 91
\$lt 89
\$lte 89
\$ne 89
\$near and \$geoNear
differences 263
\$near query 264
\$or operator 88
:as option 167
@author instance variable 221
@authors array 232
:autosave option 167, 168
:cascade_callbacks option 175
:cascaded_callbacks option 167
:class_name option 166
:cyclic option 167, 175
:dependent option 167
about 168
values 168
:embeds_one, options
about 175
:cascade_callbacks option 175
:cyclic 175
:foreign_key option 167, 168
:index option 167, 169

:inverse_of option 166, 170
:name option 166, 177
:order option 167, 168
:polymorphic option 167, 169
--replicaSet option 273
:versioned option 167, 176

A

accepts_nested_attributes_for method 236
ACID transactions and MongoDB transactions
selecting between 77
active/passive mode 266
ActiveSupport 233
Address model
geocoding 255, 256
Aeroplane model 130
AeroSpace 125
all method 113
Apdex 201
Apdex Score, server performance 201
 ApplicationController 217
Application Performance Index. *See Apdex*
arbiters 271
arrays
searching in 90
arrays and hashes
embedded objects 165
using, in models 164
atomic updates 75
attributes, in models
accessing 158
defining 157

dynamic fields 160
 indexing 158
 localization 162
author: charles 118
Author class
 modeling 210
author document 50
author_id field 118
Author object 219
AuthorsController
 about 217
 models, relating 220-222
 N+1 query problem, solving 219, 220
 writing 218, 219
Authors listing page
 authors, listing 231-234
 books, adding 234-239
 designing 231
 new authors, adding 234-239
average response time, server performance 200

B

basic embedded polymorphism, embedded polymorphism
 about 142
 drivers, insuring 142, 143
Basic polymorphic relations
 about 128
 selecting 132
 vehicles, creating 129, 131
belongs_to 118
belongs_to, options
 :index option 169
 :polymorphic option 169
 about 169
be_valid 245
Binary JSON (BSON)
 about 21, 70, 100
 data, fetching 71
 data, manipulating 71
 data, traversing 71
blueprint template 238
BookDetail model 121
BookDetail object 123
book model
 building 48-51
 writing 211

book object 92
 creating 32
BSON data
 fetching 71
 manipulating 71
 traversing 71
bsondump 28
bson_ext gem
 about 204
 used, for increasing Mongoid performance 204
Bundler
 about 44
 need for 44

C

caching objects
 about 205
 memcache server, using 205
 Redis server, using 205
capped collections 72
CarDriver object 128
Car model 131
Category model 212
category object 93
changes, in models
 managing 178
code documentation
 YARD used 247, 248
code optimization
 about 202
 data selection, optimizing 203
 indexing fields 202
collections, MongoDB
 about 72
 capped collections 72
common options, relations
 :class_name 165
 :extend 165
 :inverse_class_name 165
 :inverse_of 166
 :name 166
 :relation 166
 :validate 166
Compare and Set (CAS) 75
concurrency/throughput, server performance 201

concurrent requests 198
conditional queries
 \$exists, using 89
 books, finding by name or publisher 88
 highly ranked books, finding 89
 threshold queries, writing 88
 writing 87
 writing, \$or operator used 88
config/mongoid.yml file 149
config server
 starting 285, 286
configuration parameters, find() query
 fields 83
 imit 83
 query 83
 skip 83
covered indexes
 about 193
 using 193-195
create method 220
criteria 113
Cross Site Request Forgery (CSRF) 218
cyclic relations
 setting up 175, 176

D

data mapper 99, 100
data searching
 searching by field attributes 81
 searching by string value 82
 searching inside arrays 90
 searching inside embedded documents 93
 searching inside hashes 92
 searching with regular expressions 93
 techniques 81
dates, MongoDB 72
describe 245
document relations
 creating 37, 38
document relationships
 using 36
documents
 about 71
 creating 32, 33, 110
 creating, NoSQL way 33
 creating, SQL way 33

destroying 110
fields, defining using Mongoid 111
fields, defining using MongoMapper 110
objects, creating 111
objects, updating 111, 112
 updating 110

Don't Repeat Yourself(DRY) principle 216

Driver model 125

dynamic fields
 about 160
 adding 160, 161

E

e-mail address
 validating 96
embedded documents
 about 75
 searching in 93
 using 34-57
embedded_in, options
 about 176
 :name option 177
embedded objects
 adding, to book 35
 creating 134
 fetching 36
 Mongoid, using 134-137
 MongoMapper, using 134, 137
 using 133
embedded polymorphic relations 177
embedded polymorphism
 about 140
 basic embedded polymorphism 142
 Single Collection Inheritance 141
embeds_many, options
 :versioned option 176
 about 176
emit() 63
end-user response 202
exact matches
 searching for, \$all used 92
explain function
 about 190
 query, explaining 190-193
 using 190
extend 49

F

failover 266
field attributes
 searching by 81
fields
 localizing 162, 163
finder methods
 all method, using 113
 find method 112
 first and last methods, using 113
 using 112
finders 112
find method 113
find() query
 about 83
 configuration parameters 83
following and followers relationship
 configuring 172-174
functional programming 40

G

gemset 17
geo 252
geocoder
 used, for updating geolocation coordinates 258, 259
geocoder gem 259
Geographical Information Systems(GIS) 252
geolocation
 about 252
 accuracy 253
 converting, to geocoded coordinates 253
 identifying 254, 255
geolocation coordinates
 saving 257
 updating, geocoder used 258, 259
geolocation queries
 \$near and \$geoNear, differences 263, 264
 about 260
 mongoid_spacial, using 262
 nearby addresses, finding 260-262
 near queries, firing in Mongoid 262, 263
geolocation storage
 testing 257
geospatial indexes
 adding, to MongoDB 255

geospatial indexing 251
global write lock 75
GROUP BY query 64

H

has_and_belongs_to_many, options
 :inverse_of option 170
 about 169
hashes
 searching in 92
has_many 118
has_many, options
 :order option 168
 about 168
has_one, options
 :as option 167
 :autosave option 168
 :dependent option 168
 :foreign_key option 168
high availability 266
highly ranked books
 finding 89
Horizontal scaling 265
httpperf
 used, for loading server 198, 199

I

include 49
includes 219
indexing attributes
 about 158
 background indexing 159
 geospatial indexing 159
 sparse indexing 160
 unique indexes 159
initiate command 274
interleaving 75
Internationalization 162
it 245

J

JavaScript
 about 72, 73
 and, MongoDB 72
 custom functions, writing in MongoDB 73

JavaScript Object Notation. See **JSON**
JSON 21

L

Lease and Purchase models
embedding 58, 59
Lease model
writing 213
Localization 162
local.slaves collection 270
location 252

M

many 118
many-to-many relation
about 56, 118
accessing, with Mongoid 120, 121
accessing, with MongoMapper 120
books, categorizing 118
configuring 171, 172
Mongoid, using 119
MongoMapper, using 118, 119
map function
about 40, 292
building 40
writing, for calculating ratings 63
writing, for calculating vote statistics 41
Map/Reduce
about 40
using 64
working with 60-63
working with, Ruby used 65
mapReduce function 292
Map/Reduce functionality
implementing 289
Map/Reduce functionality planning 290
Map/Reduce via mongo console 291, 292
Map/Reduce via Ruby 293, 294
Marine 125
Marine object 128
master/slave replication
setting up 266-271
memcache server
setting up 205
memory-mapped storage engine
performance 203

using 74
Metal 150
model relationships
about 116
many-to-many relation 118
one to many relation 116
one-to-one relation 121
polymorphic relations 124
model, Ruby
book model, building 48
building 48
object schema, planning 48
remaining models, building 51, 52
Model-View-Controller (MVC) architecture 215
module mixin 49
mongo 22
Mongo::Connection class 103
MongoDB
and, JavaScript 72
backup, managing using mongodump 25
code, optimizing 202
collections 72
comparing, with SQL syntax 38, 39
configuring 19
connecting, mongo used 22
covered indexes 193
data, importing using mongoimport 25
data searching 81
dates 72
document relations, creating 37, 38
document relationships, using 36
documents 71
documents, creating 32, 33
embedded documents, using 34
embedded objects, adding to book 35
embedded objects, fetching 36
explain function 190
files, saving using mongofiles 26
functional programming 40
geolocation queries, firing 260
geospatial indexes, adding 255
geospatial indexing 251
global write lock 74
information, deleting 24
information, exporting using mongoexport 24
information, retrieving 23
information, saving 22

installing 18
limitations 77
many-to-many relationships 56
map function, building 40
Map/Reduce, using 40
master/slave replication, implementing 266
memory-mapped storage engine, using 74
performance tuning techniques 196
profiling 188
profiling, enabling 188, 189
reduce function, building 41
replica sets 271
replication schemes 266
restore, managing using mongorestore 25
reviews and votes, embedding 35
Ruby DataMappers 103
starting 19, 20
stopping 21
storing coordinates 255
transactional support 75
web application performance 197
web application stack, optimizing 203
web application stack, tuning 203
write-ahead journaling 74
write consistency, ensuring 73

MongoDB CLI
about 21
bsondump 28
JSON 21
mongo client utility 22
mongodump 25
mongoexport 24
mongofiles 26
mongoimport 25
mongorestore 25

MongoDB criteria
conditional queries, executing using where 113
limit 115
offset 115
results, fetching with where criteria 114
skip 115
using 113
where criteria, using for fetching results 114

Mongo::Db object 103

Mongo driver
configuration 102

mongodump

used, for managing backup 25

mongoexport
used, for exporting information 24

mongofiles
used, for saving files 26

mongo gem
installing 100
using 100

Mongoid
about 46, 104
arrays and hashes, using 164
attributes, defining 157
changes, managing 178
configuring 47, 107, 109, 110
relations, defining 165
reverse embedded relations 137
setting up 46
web application, developing 147

Mongoid::Criteria object 114

Mongoid::Document
field method 157
optional arguments 157

Mongoid modules
about 179
Paranoia module 180
versioning 182

mongoid_spacial
using 262

mongoimport
used, for importing information 25

MongoMapper
about 104
configuring 104, 105
used, for creating models 106

MongoMapper::Document
about 106
modules 109
plugins 108

mongorestore
used, for managing restore 25

mongo-ruby-driver
about 100
mongo gem, using 101, 102

mongos process
routing service, setting up 286
setting up 286-288

mongostat 197

Mongrel 204
MRI Ruby 12

N

nested_form method 238
network latency 202
NoSQL scores
 over, SQL databases 33
NoSQL way 33

O

Object Document Mapper (ODM) tool 46
ObjectId 71
Occurrence 95
one to many relation
 about 116
 models, relating 116
 Mongoid, using 117, 118
 MongoMapper, using 116
one-to-one relation
 about 121
 book details, adding 123
 models, creating 124
 Mongoid, using 122
 MongoMapper, using 122
optimistic locking
 implementing 75, 76
optional arguments, Mongoid::Document
 :as 157
 :default 157
 :identity 157
 :localize 157
 :type 157
Order model
 writing 212

P

Paranoia module
 about 180, 181
 including 180, 181
Pattern 95
people criterion 115
performance benchmarking
 about 295
 Ruby objects, iterating 295, 296

performance tuning techniques
 about 196
 mongostat 197
Pilot object 128
Polymorphic 124
polymorphic relations
 about 124
 implementing, correct way 124
 implementing, wrong way 124
polymorphic relations, implementing
 Basic polymorphic relations 128
 Single Collection Inheritance (SCI) 124
PRIMARY node 274
profiling
 about 188
 enabling, for MongoDB 188, 189
protect_from_forgery 218
Purchase model
 writing 213

R

Rack 156
Rails
 about 44, 208
 Author class, modeling 210
 Authors listing page, designing 231
 basics 44
 components 208
 Controllers, coding 217
 project, setting up 208, 209
 Rails architecture 215
 Rails request, processing 216, 217
 Rails routes 213
 RESTful interface 214
 Sodibee, modeling 210
 Views, coding 217
 web application layout, designing 223

Rails 3
 about 28, 148
 installing 28
Rails application
 setting up 148, 149
Rails architecture 215, 216
Rails asset pipeline 230
Rails ORM 48

Rails project
 creating 43
 setting up 43, 208, 209
 testing 52-55

Rails request
 processing 216, 217

Rails/Sinatra
 installing 28

railtie 148

rake routes command 216

rbenv
 about 17
 used, for installation Ruby 17

reactor pattern 198

Redis server 205

reduce function
 about 41, 64, 292
 building 41
 writing, for processing emitted information 42,
 43
 writing, for processing emitted results 64

regular expressions
 Occurrence 95
 Pattern 95
 searching 93
 searching with 93

regular expression searches
 using 94

relations, in models
 :embeds_one, options 175
 belongs_to, options 169
 common options 165
 defining 165
 embedded_in, options 176
 embeds_many, options 176
 has_and_belongs_to_many, options 169
 has_many, options 168
 has_one, options 167
 relation-specific options 166

relation-specific options
 :as 167
 :autosave 167
 :cascaded_callbacks 167
 :cyclic 167
 :dependent 167
 :foreign_key 167
 :index 167

:order 167
 :polymorphic 167
 :versioned 167

replica sets
 about 271
 configuring, for Sodibee 278-281
 implementing 272-277
 implementing, for Sodibee 278
 members, adding 277

replication 266

resource_id field 131

resource_type field 131

REST 213

RESTful interface
 about 214
 routes, configuring 214, 215

reverse embedded relations
 about 137
 embeds_many, using 139, 140
 embeds_one relationship, using 138, 139

review_count field 34

reviews
 adding, to books 57, 58
 embedding 35
 searching in 90

routing service
 setting up 286, 287

RSpec
 about 244
 basics 245
 be_valid 245
 describe 245
 installing 244, 245
 it 245
 should 245
 should_not 245
 spork, installing 246
 used, for automation 243
 used, for testing 243

rs.slaveOk() 277

rs.status() command 275

Ruby
 about 12
 Bundler, using 44
 installing 12
 installing, RVM used 12
 models, building 48

Rails project, setting up 43
requisites 11
Sodibee, setting up 45

Ruby application server
Mongrel 204
passenger 204
selecting 204
Thin 204
Unicorn 204

Ruby DataMappers
about 103
embedded objects, using 133
features 99
finder methods, using 112
Mongoid 103
Mongoid, configuring 107
MongoMapper 103
MongoMapper, configuring 104
need for 99
setting up 104

Ruby installation
about 12
rbenv, used 17
RVM games 16
RVM, installing 12
RVM packages, configuring 15
RVM, using on Linux or Mac OS 12, 14, 16
Windows saga 17

Ruby Version Manager. *See RVM*

RVM
about 12
using, on Linux or Mac OS 12, 15

RVM games 16

S

searching by field attributes, data searching
about 81, 82
conditional queries, writing 87
document results, paginating 87
documents, skipping 86, 87
fields, excluding 86
fields, including 86
searching by string value 82, 83
search results, limiting 86, 87
skip and limit, using 86
specific fields, querying for 84, 85

searching inside arrays, data searching
\$in and \$nin, used 91
about 90
exact matches, searching for 92
searching inside reviews 90, 91

searching inside embedded documents, data searching 93

searching inside hashes, data searching 92

searching with regular expressions, data searching

about 93-95
e-mail address, validating 96

sharding

about 283
implementing 283

shards

configuring, with config server 285, 286
creating 284
setting up 284, 285

shared replication

shared replication testing 288, 289

shelf collection 32

shims 17

ShipDriver object 128

Ship model 129

should 245

should_not 245

simple_form method 237

Sinatra

about 240
installing 28
setting up 149, 150, 240-243
using, professionally 151-156

Single Collection Inheritance, embedded polymorphism

about 141
licenses, adding to drivers 141

Single Collection Inheritance (SCI)

about 125
driver entities, managing 125-128
hierarchy 125
selecting 132

Sodibee

replica sets, implementing 278-280

Sodibee project

Address model, writing 212
Author class, modeling 210

Book model, writing 211
Category model, writing 212
modeling 210
Mongoid, configuring 47
Mongoid, setting up 46
Order model, modeling 212, 213
revisiting 208
setting up 45
SpaceShuttle model 130
specific fields
 querying for 84, 85
spork
 installing 246
SQL way 33
storing coordinates
 about 255
 Address model, geocoding 255, 256
 geolocation storage, testing 257
Submarine model 130

T

Terrestrial 125
Thin 204
threshold queries
 writing 88
throughput
 about 198
 server, loading using httpperf 198, 199
 server performance, monitoring 199, 200
time to live(TTL) 205
to_sentence method 233
transactional support, MongoDB
 atomic updates 75
 embedded documents 75
 optimistic locking, implementing 75

U

Unicorn 204

V

Vehicle model 129
Versioning module
 about 182, 183
 including 182, 183
Vertical scaling 265

vote_count field 34

votes
 embedding 35
votes array 66

W

web application
 developing, with Mongoid 147
web application layout
 designing 223
 layout, designing 223-230
 Rails asset pipeline 230
web application performance
 about 197
 end-user response 202
 network latency 202
 standard parameters 197
 throughput 198
 web server response time 197
web application stack optimization
 caching objects 205
 memory-mapped storage engine
 performance 203
 Mongoid performance, increasing 204
 optimizing 203
 Ruby application server, selecting 204
web server
 loading, httpperf used 198, 199
web server performance
 Apdex Score 201
 average response time 200
 concurrency/throughput 201
 monitoring 199, 200
web server response time 197
Windows saga 17
write-ahead journaling
 about 74
 advantages 74
write consistency
 ensuring 73

Y

YARD
 about 247
 installing 247, 248



Thank you for buying **Ruby and MongoDB Web Development Beginner's Guide**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

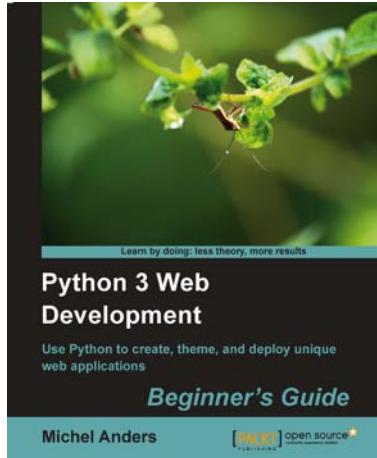
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Python 3 Web Development Beginner's Guide

ISBN: 978-1-84951-374-6 Paperback: 336 pages

Use Python to create, theme, and deploy unique web applications

1. Build your own Python web applications from scratch
2. Follow the examples to create a number of different Python-based web applications, including a task list, book database, and wiki application
3. Have the freedom to make your site your own without having to learn another framework
4. Part of Packt's Beginner's Guide Series: practical examples will make it easier for you to get going quickly



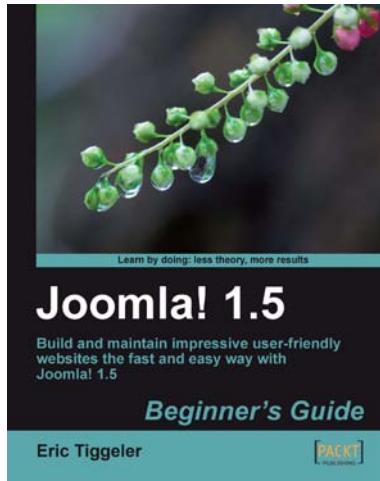
Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 450 pages

Over 130 easy to follow recipes backed up with real life examples, walking you through the basic Ext JS features to advanced application design using Sencha Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

Please check www.PacktPub.com for information on our titles

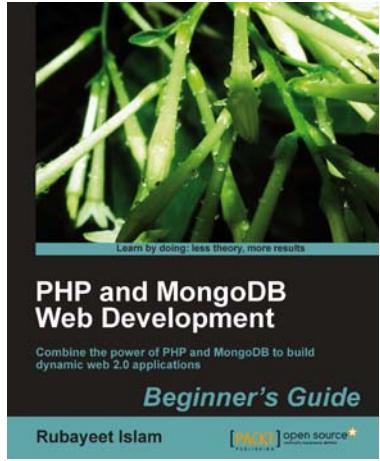


Joomla! 1.5: Beginner's Guide

ISBN: 978-1-847199-90-4 Paperback: 380 pages

Build and maintain impressive user-friendly web sites the fast and easy way with Joomla! 1.5

1. Create a web site that meets real-life requirements by following the creation of an example site with the help of easy-to-follow steps and ample screenshots
2. Practice all the Joomla! skills from organizing your content to completely changing the site's looks and feel
3. Go beyond a typical Joomla! site to make the site meet your specific needs



PHP and MongoDB Web Development Beginner's Guide

ISBN: 978-1-84951-362-3 Paperback: 292 pages

Combine the power of PHP MongoDB to build dynamic web 2.0 applications

1. Learn to build PHP-powered dynamic web applications using MongoDB as the data backend
2. Handle user sessions, store real-time site analytics, build location-aware web apps, and much more, all using MongoDB and PHP
3. Full of step-by-step instructions and practical examples, along with challenges to test and improve your knowledge

Please check www.PacktPub.com for information on our titles