

THE EXPERT'S VOICE® IN DATABASES

Pro MongoDB™ Development

Deepak Vohra

Apress®

www.allitebooks.com

Pro MongoDBTM Development



Deepak Vohra

Apress®

Pro MongoDB™ Development

Copyright © 2015 by Deepak Vohra

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-1599-9

ISBN-13 (electronic): 978-1-4842-1598-2

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Steve Anglin

Technical Reviewers: Manuel Jordan, John Yeary, and Massimo Nardone

Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson,

Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper,

Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss

Coordinating Editor: Mark Powers

Copy Editor: Karen Jameson

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484215999. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Contents at a Glance

About the Author	xiii
About the Technical Reviewers	xv
Introduction	xvii
■ Chapter 1: Using a Java Client with MongoDB	1
■ Chapter 2: Using the Mongo Shell	39
■ Chapter 3: Using MongoDB with PHP	89
■ Chapter 4: Using MongoDB with Ruby	129
■ Chapter 5: Using MongoDB with Node.js	177
■ Chapter 6: Migrating an Apache Cassandra Table to MongoDB	243
■ Chapter 7: Migrating Couchbase to MongoDB	273
■ Chapter 8: Migrating Oracle Database	297
■ Chapter 9: Using Kundera with MongoDB	305
■ Chapter 10: Using Spring Data with MongoDB	345
■ Chapter 11: Creating an Apache Hive Table with MongoDB	405
■ Chapter 12: Integrating MongoDB with Oracle Database in Oracle Data Integrator	427
Index.....	477

Contents

About the Author	xiii
About the Technical Reviewersxv
Introduction	xvii
■ Chapter 1: Using a Java Client with MongoDB	1
Setting Up the Environment	1
Creating a Maven Project	2
Creating a BSON Document.....	11
Using a Model to Create a BSON Document	16
Getting Data from MongoDB.....	22
Updating Data in MongoDB	26
Deleting Data in MongoDB	32
Summary.....	37
■ Chapter 2: Using the Mongo Shell	39
Getting Started	39
Setting Up the Environment.....	39
Starting the Mongo Shell.....	40
Running a Command or Method in Mongo Shell	42
Using Databases.....	44
Getting Databases Information.....	44
Creating a Database Instance.....	46
Dropping a Database	47

■ CONTENTS

Using Collections	48
Creating a Collection	48
Dropping a Collection	52
Using Documents	53
Adding a Document	53
Adding a Batch of Documents	56
Saving a Document.....	60
Updating a Document.....	68
Updating Multiple Documents	71
Finding One Document	74
Finding All Documents.....	75
Finding Selected Fields	76
Using the Cursor	80
Finding and Modifying a Document.....	82
Removing a Document	85
Summary.....	87
■ Chapter 3: Using MongoDB with PHP	89
Getting Started	89
Overview of the PHP MongoDB Database Driver	89
Setting Up the Environment.....	91
Installing PHP	92
Installing PHP Driver for MongoDB	93
Creating a Connection	94
Getting Database Info	97
Using Collections	98
Getting a Collection	99
Dropping a Collection	101
Using Documents	102
Adding a Document	102
Adding Multiple Documents.....	106
Adding a Batch of Documents	107

Finding a Single Document.....	110
Finding All Documents.....	112
Finding a Subset of Fields and Documents	114
Updating a Document.....	117
Updating Multiple Documents	120
Saving a Document.....	122
Removing a Document	125
Summary.....	128
■ Chapter 4: Using MongoDB with Ruby.....	129
Getting Started	129
Overview of the Ruby Driver for MongoDB	129
Setting Up the Environment.....	131
Installing Ruby	131
Installing DevKit.....	133
Installing Ruby Driver for MongoDB.....	134
Using a Collection.....	136
Creating a Connection with MongoDB	136
Connecting to a Database.....	139
Creating a Collection	143
Using Documents	147
Adding a Document	147
Adding Multiple Documents.....	151
Finding a Single Document.....	156
Finding Multiple Documents.....	158
Updating Documents	162
Deleting Documents	169
Performing Bulk Operations	173
Summary.....	176

■ Chapter 5: Using MongoDB with Node.js	177
Getting Started	177
Overview of Node.js Driver for MongoDB	177
Setting Up the Environment.....	178
Installing MongoDB Server.....	178
Installing Node.js	178
Installing the Node.js Driver for MongoDB.....	185
Using a Connection	186
Creating a MongoDB Connection.....	186
Using the Database	191
Using a Collection.....	198
Using Documents	205
Adding a Single Document	205
Adding Multiple Documents.....	208
Finding a Single Document.....	210
Finding All Documents.....	213
Finding a Subset of Documents.....	215
Using the Cursor	217
Finding and Modifying a Single Document.....	221
Finding and Removing a Single Document.....	224
Replacing a Single Document.....	227
Updating a Single Document	229
Updating Multiple Documents	233
Removing a Single Document.....	235
Removing Multiple Documents.....	237
Performing Bulk Write Operations	239
Summary.....	242

■ Chapter 6: Migrating an Apache Cassandra Table to MongoDB	243
Setting Up the Environment	243
Creating a Maven Project in Eclipse	245
Creating a Document in Apache Cassandra	255
Migrating the Cassandra Table to MongoDB	263
Summary	272
■ Chapter 7: Migrating Couchbase to MongoDB.....	273
Setting Up the Environment	273
Creating a Maven Project	274
Creating Java Classes	278
Configuring the Maven Project	281
Adding Documents to Couchbase	283
Creating a Couchbase View.....	288
Migrating Couchbase Documents to MongoDB.....	292
Summary	296
■ Chapter 8: Migrating Oracle Database	297
Overview of the mongoimport Tool.....	297
Setting Up the Environment	298
Creating an Oracle Database Table	299
Exporting an Oracle Database Table to a CSV File.....	299
Importing Data from a CSV File to MongoDB.....	301
Displaying the JSON Data in MongoDB	302
Summary	304
■ Chapter 9: Using Kundera with MongoDB	305
Setting Up the Environment	306
Creating a MongoDB Collection.....	306
Creating a Maven Project in Eclipse	307
Creating a JPA Entity Class	312

Configuring JPA in the persistence.xml Configuration File	318
Creating a JPA Client Class	325
Running CRUD Operations.....	327
Creating a Catalog	328
Finding a Catalog Entry Using the Entity Class.....	328
Finding a Catalog Entry Using a JPA Query	329
Updating a Catalog Entry.....	330
Deleting a Catalog Entry	330
The Kundera-Mongo JPA Client Class	331
Installing the Maven Project.....	333
Running the Kundera-Mongo JPA Client Class	338
Invoking the KunderaClient Methods.....	339
Summary.....	343
Chapter 10: Using Spring Data with MongoDB	345
Setting Up the Environment	345
Creating a Maven Project.....	346
Installing Spring Data MongoDB	350
Configuring JavaConfig	351
Creating a Model	353
Using Spring Data MongoDB with Template.....	354
Creating a MongoDB Collection	356
Creating Document Instances.....	358
Adding a Document	358
Adding a Document Batch	361
Finding a Document by Id	363
Finding One Document	364
Finding All Documents	368
Finding Documents Using a Query	370

Updating the First Document.....	373
Update Multiple Documents	376
Removing Documents.....	379
Using Spring Data with MongoDB	389
Getting Document Count.....	391
Finding Entities from Repository	392
Saving Entities.....	395
Deleting Entities	399
Deleting a Document By Id	399
Deleting All Documents	400
Summary.....	404
■ Chapter 11: Creating an Apache Hive Table with MongoDB	405
Overview of Hive Storage Handler for MongoDB.....	405
Setting Up the Environment	406
Creating a MongoDB Data Store.....	409
Creating an External Table in Hive.....	422
Summary.....	426
■ Chapter 12: Integrating MongoDB with Oracle Database in Oracle Data Integrator	427
Setting Up the Environment	427
Creating the Physical Architecture	430
Creating the Logical Architecture	443
Creating the Data Models.....	447
Creating the Integration Project	459
Creating the Integration Interface	464
Running the Interface.....	470
Selecting Integrated Data in Oracle Database Table	476
Summary.....	476
Index.....	477

About the Author



Deepak Vohra is a consultant and a principal member of the [NuBean.com](#) software company. Deepak is a Sun-certified Java programmer and Web component developer. He has worked in the fields of XML, Java programming, and Java EE for over ten years. Deepak is the author of *Pro Couchbase Development* (Apress, 2015) and the coauthor of *Pro XML Development with Java Technology* (Apress, 2006). Deepak is also the author of the *JDBC 4.0* and *Oracle JDeveloper for J2EE Development*, *Processing XML Documents with Oracle JDeveloper 11g*, *EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*, and *Java EE Development in Eclipse IDE* (Packt Publishing). He also served as the technical reviewer on *WebLogic: The Definitive Guide* (O'Reilly Media, 2004) and *Ruby Programming for the Absolute Beginner* (Cengage Learning PTR, 2007).

About the Technical Reviewers



Manuel Jordan is an autodidactic developer and researcher who enjoys learning new technologies so that he can experiment with new ways to integrate them.

Manuel won the 2010 Springy Award – Community Champion and Spring Champion 2013. In his little free time, he reads the Bible and composes music on his bass and guitar.



Massimo Nardone holds a Master of Science degree in Computing Science from the University of Salerno, Italy. He worked as a PCI QSA and Senior Lead IT Security/Cloud/SCADA Architect for many years and currently works as Security, Cloud and SCADA Lead IT Architect for Hewlett Packard Finland. He has more than twenty years of work experience in IT including Security, SCADA, Cloud Computing, IT Infrastructure, Mobile, Security, and WWW technology areas for both national and international projects. Massimo has worked as a Project Manager, Cloud/SCADA Lead IT Architect, Software Engineer, Research Engineer, Chief Security Architect, and Software Specialist. He worked as visiting a lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). Massimo has been programming and teaching how to program with Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than twenty years. He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo is the author of *Pro Android Games* (Apress, 2015).

Introduction

MongoDB server is ranked first in NoSQL databases and fourth in all databases (relational or NoSQL). While several books on MongoDB administration are available, none on MongoDB-based development are available.

This Pro *MongoDB™ Development* book is about MongoDB server, a NoSQL database based on the BSON (binary JSON) document model. As already noted, MongoDB is the most commonly used NoSQL database and is ranked fourth in all databases (relational or NoSQL) according to DB-Engines.com (Reference: <http://db-engines.com/en/ranking>). The book discusses all aspects of using MongoDB database in web development. Java, PHP, Ruby, and JavaScript are the most commonly used programming/scripting languages, and the book discusses accessing MongoDB database with these languages. The book also discusses using Java EE frameworks Kundera and Spring Data with MongoDB. As NoSQL databases are commonly used with the Hadoop ecosystem, the book discusses using MongoDB with Apache Hadoop and Apache Hive. An Oracle Data Integrator-based integration of MongoDB data into Oracle Database is also discussed. Migration from other NoSQL databases (Apache Cassandra and Couchbase) and from relational database (Oracle Database) is discussed.

This book is for web developers and NoSQL developers who develop applications using MongoDB server. Prerequisite knowledge of Java, Java EE, and scripting languages PHP, Ruby, and JavaScript is essential. Familiarity with Apache Hadoop, Apache Hive, Oracle Database, and Oracle Data Integrator is also a prerequisite.

According to the Java Tools and Technologies Landscape for 2014, “In the NoSQL world... among the developers using NoSQL... MongoDB (56%) is clearly leading...” (Reference: <http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/13/>).

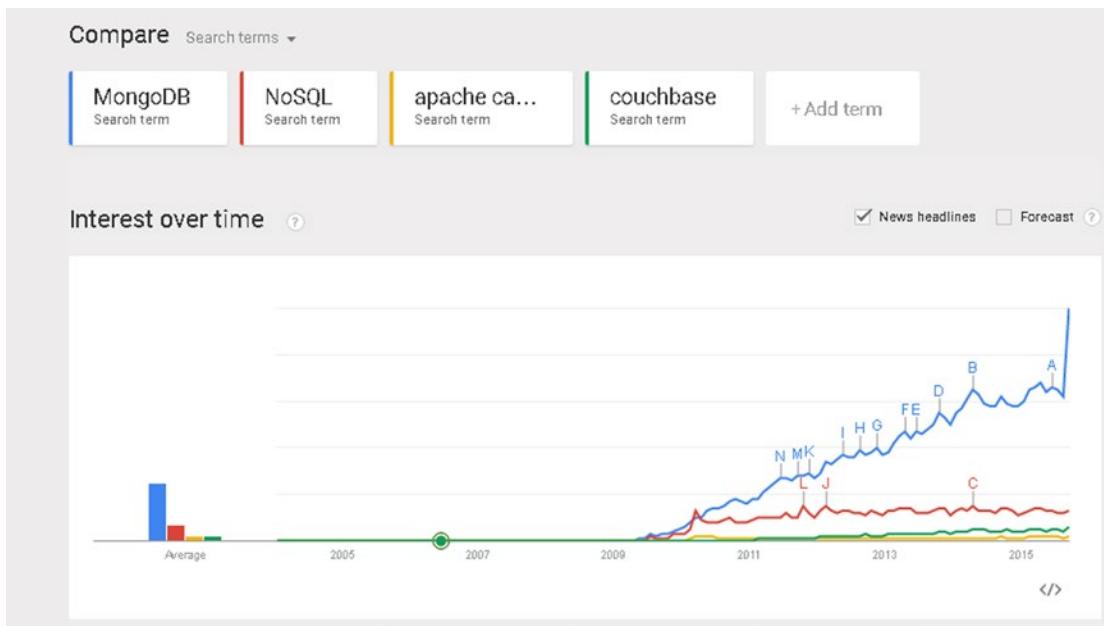
More than 2000 organizations use MongoDB (Reference: www.mongodb.com/who-uses-mongodb).

Various metrics have ranked MongoDB above all other NoSQL databases (Reference: www.mongodb.com/leading-nosql-database).

MongoDB was named as the Database of the Year in 2013 by DB-Engines (Reference: www.mongodb.com/blog/post/mongodb-named-2013-database-year-why-matters).

Google Trends search term ranking for “MongoDB” as compared with “Apache Cassandra” and “Couchbase” in September 2015 is shown in the following illustration.

■ INTRODUCTION



Google Trends Database Search Results

CHAPTER 1



Using a Java Client with MongoDB

MongoDB server provides drivers for several languages including Java. The MongoDB Java driver may be used to connect to MongoDB server from a Java application and create a collection, get a collection or a list of collections, add a document or multiple documents to a collection, and find a document or a set of documents. In this chapter we shall create a Java application in Eclipse and access MongoDB server to add a document and subsequently get data from MongoDB server and also update and delete data in MongoDB server. This chapter covers the following topics:

- Setting up the environment
- Creating a Maven project
- Creating a BSON document in MongoDB
- Using a model to create a BSON document in MongoDB
- Getting data from MongoDB
- Updating data in MongoDB
- Deleting data in MongoDB

Setting Up the Environment

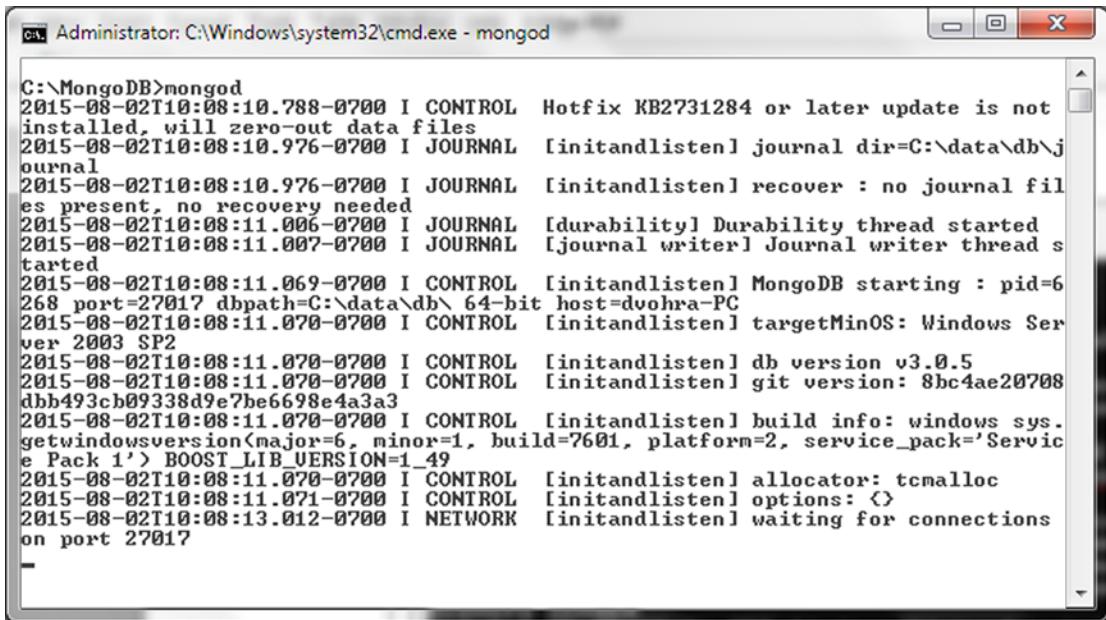
We need to download and install the following software.

1. MongoDB 3.0.5 binary distribution `mongodb-win32-x86_64-3.0.5-signed.msi` from www.mongodb.org/downloads. Stable Release (3.0.5) for Windows 64-bit is used in this chapter.
2. Eclipse IDE for Java EE Developers from www.eclipse.org/downloads.
3. Java 5 or later (Java 7 used) from www.oracle.com/technetwork/java/javase/downloads/index.html.

Double-click on the MongoDB binary distribution to install MongoDB. Add the bin directory (`C:\Program Files\MongoDB\Server\3.0\bin`) of the MongoDB installation to the PATH environment. Create a directory `C:\data\db` for the MongoDB data. Start the MongoDB server with the following command.

```
>mongod
```

The MongoDB server gets started as shown in Figure 1-1.



The screenshot shows a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongod". The window displays the log output of the MongoDB daemon (mongod) starting up. The log includes various informational messages such as control and journal initialization, durability thread startup, and network connection waiting.

```
C:\MongoDB>mongod
2015-08-02T10:08:10.788-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-08-02T10:08:10.976-0700 I JOURNAL [initandlisten] journal dir=C:\data\db\j
ournal
2015-08-02T10:08:10.976-0700 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-08-02T10:08:11.006-0700 I JOURNAL [durability] Durability thread started
2015-08-02T10:08:11.007-0700 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-08-02T10:08:11.069-0700 I CONTROL [initandlisten] MongoDB starting : pid=6
268 port=27017 dbpath=C:\data\db\ 64-bit host=dvohra-PC
2015-08-02T10:08:11.070-0700 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-08-02T10:08:11.070-0700 I CONTROL [initandlisten] db version v3.0.5
2015-08-02T10:08:11.070-0700 I CONTROL [initandlisten] git version: 8bc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-08-02T10:08:11.070-0700 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-08-02T10:08:11.070-0700 I CONTROL [initandlisten] allocator: tcmalloc
2015-08-02T10:08:11.071-0700 I CONTROL [initandlisten] options: <>
2015-08-02T10:08:13.012-0700 I NETWORK [initandlisten] waiting for connections
on port 27017
```

Figure 1-1. Starting MongoDB

Creating a Maven Project

To access MongoDB from a Java application we need to create a Maven project in Eclipse IDE and add the required dependencies.

1. Select File ▶ New ▶ Other. In the New window, select the Maven ▶ Maven Project wizard and click on Next as shown in Figure 1-2.

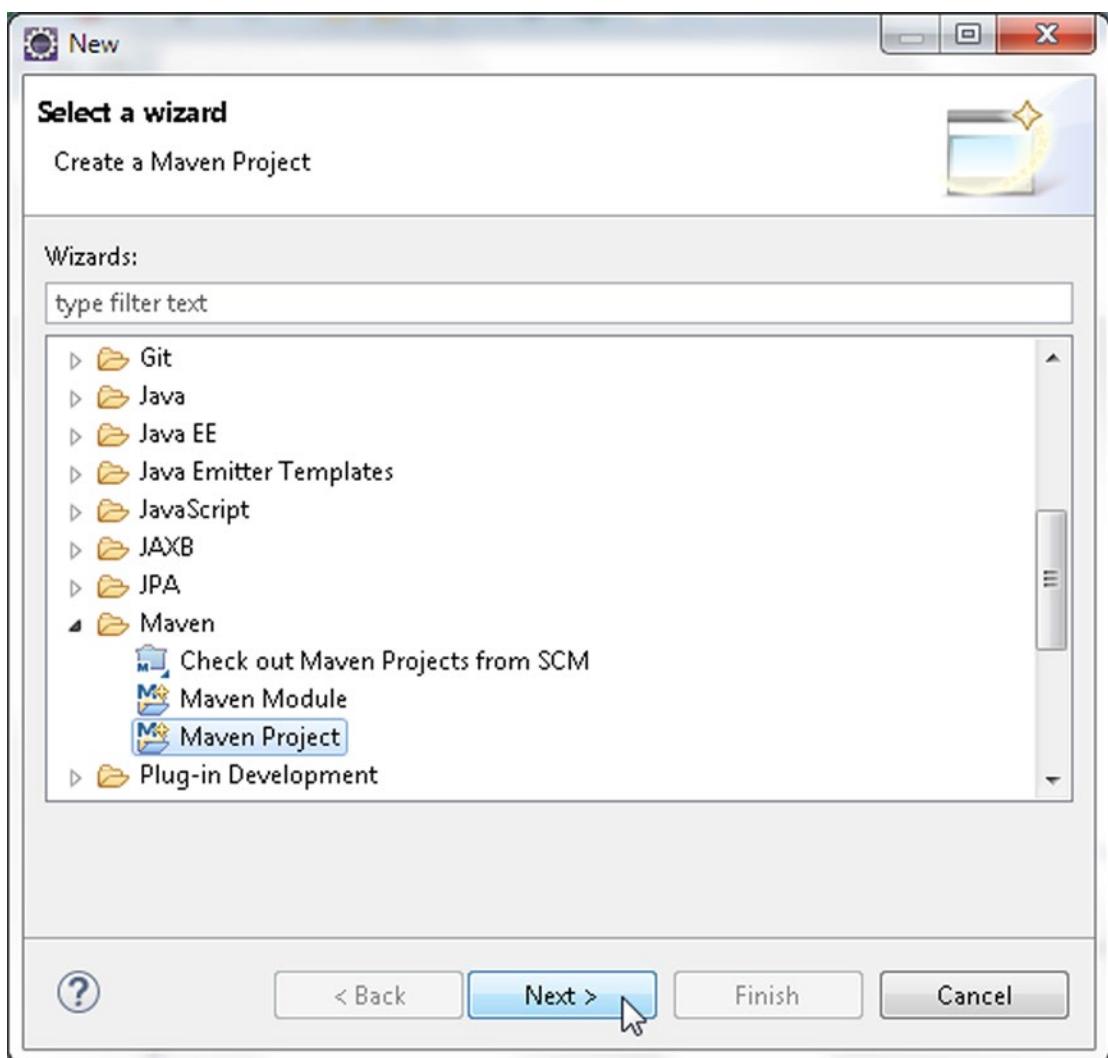


Figure 1-2. Selecting Maven ► Maven Project

2. In New Maven Project wizard select the check boxes “Create a simple project” and “Use default Workspace location” as shown in Figure 1-3. Click on Next.

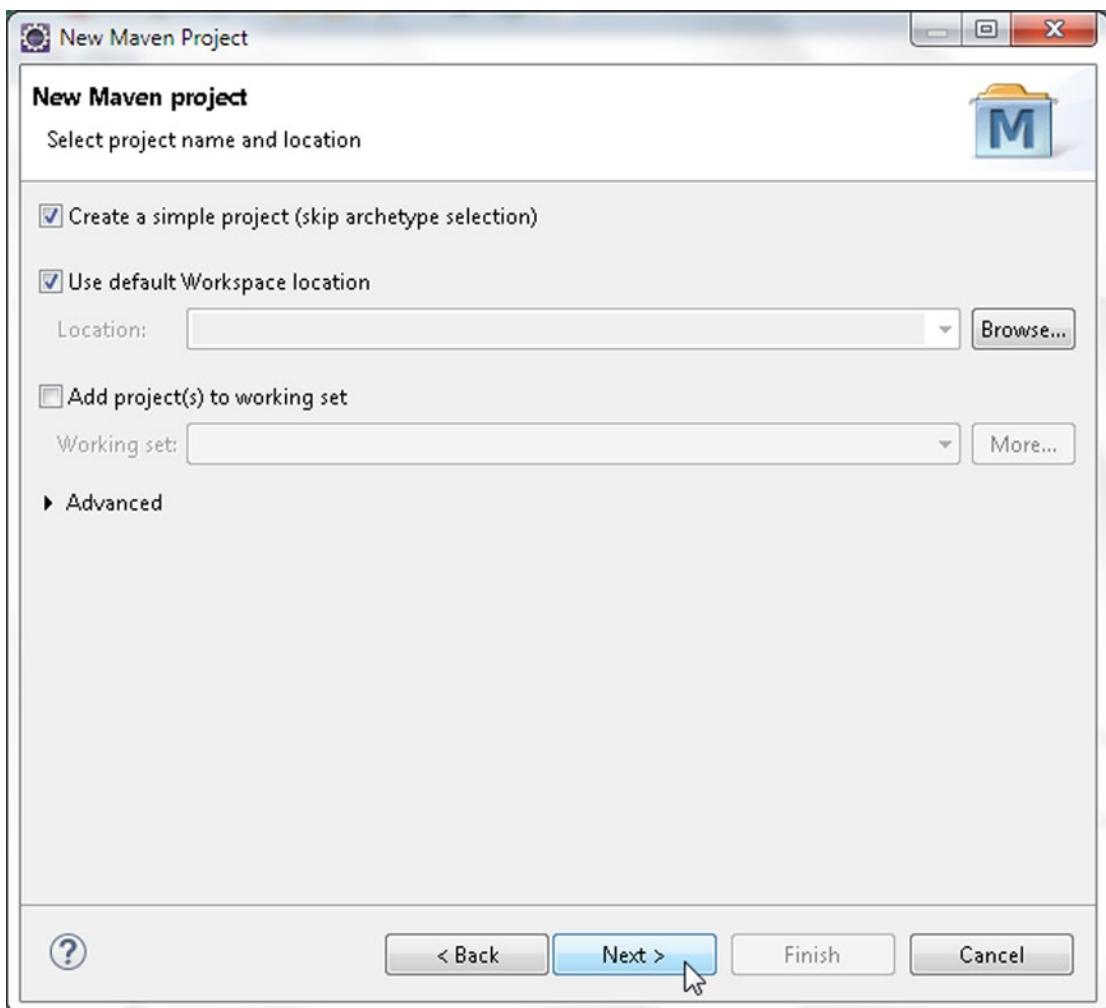


Figure 1-3. New Maven Project wizard

3. Next, configure the project, specifying the following values as shown in Figure 1-4 and then clicking on Finish:
 - Group Id: `mongodb.java`
 - Artifact Id: `MongoDBJava`
 - Version: 1.0.0
 - Packaging: jar
 - Name: `MongoDBJava`

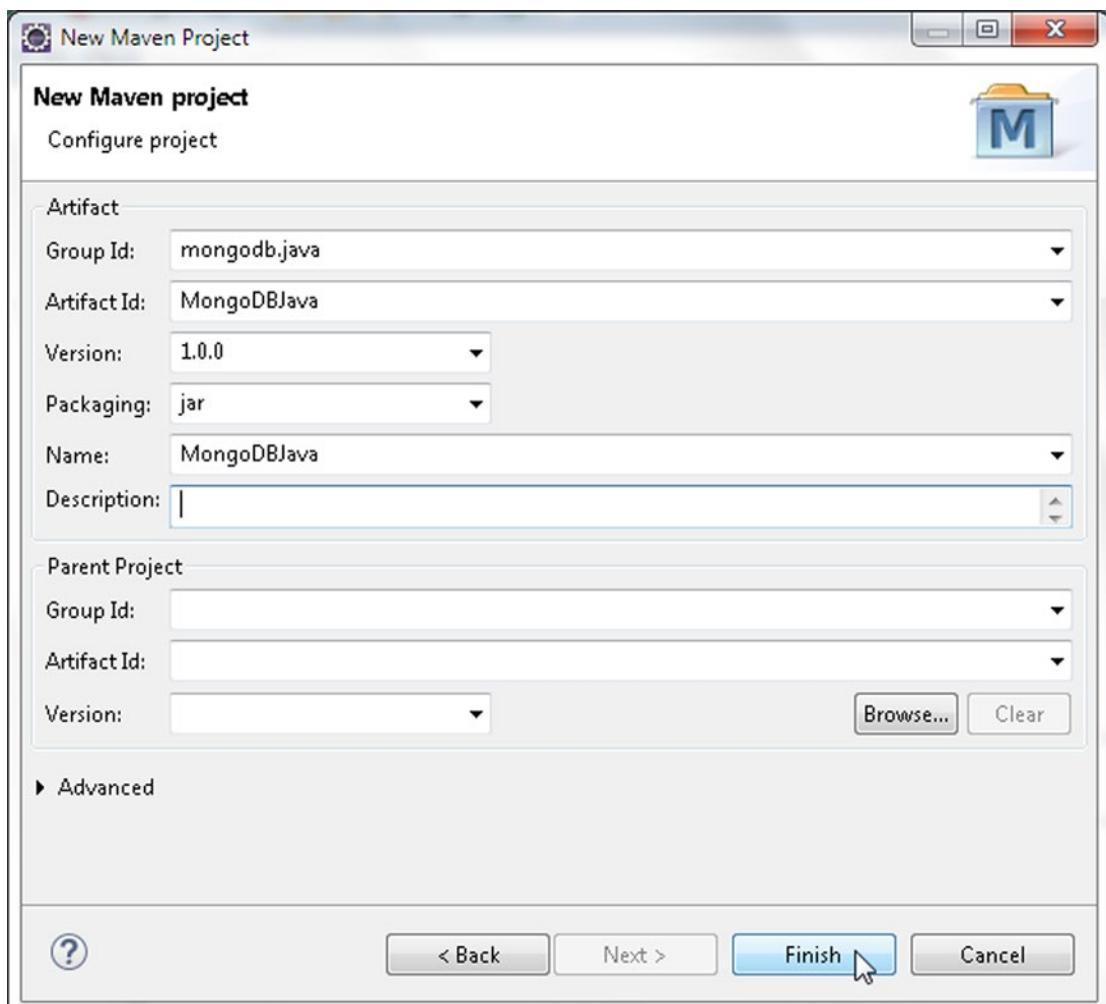


Figure 1-4. Configuring the Maven Project

A new Maven project gets created as shown in Figure 1-5.



Figure 1-5. Maven Project MongoDBJava

4. We need to add some Java classes to the Maven project to run CRUD operations on MongoDB server. Select File ► New ► Other, and in the New window, select Java ► Class as shown in Figure 1-6.

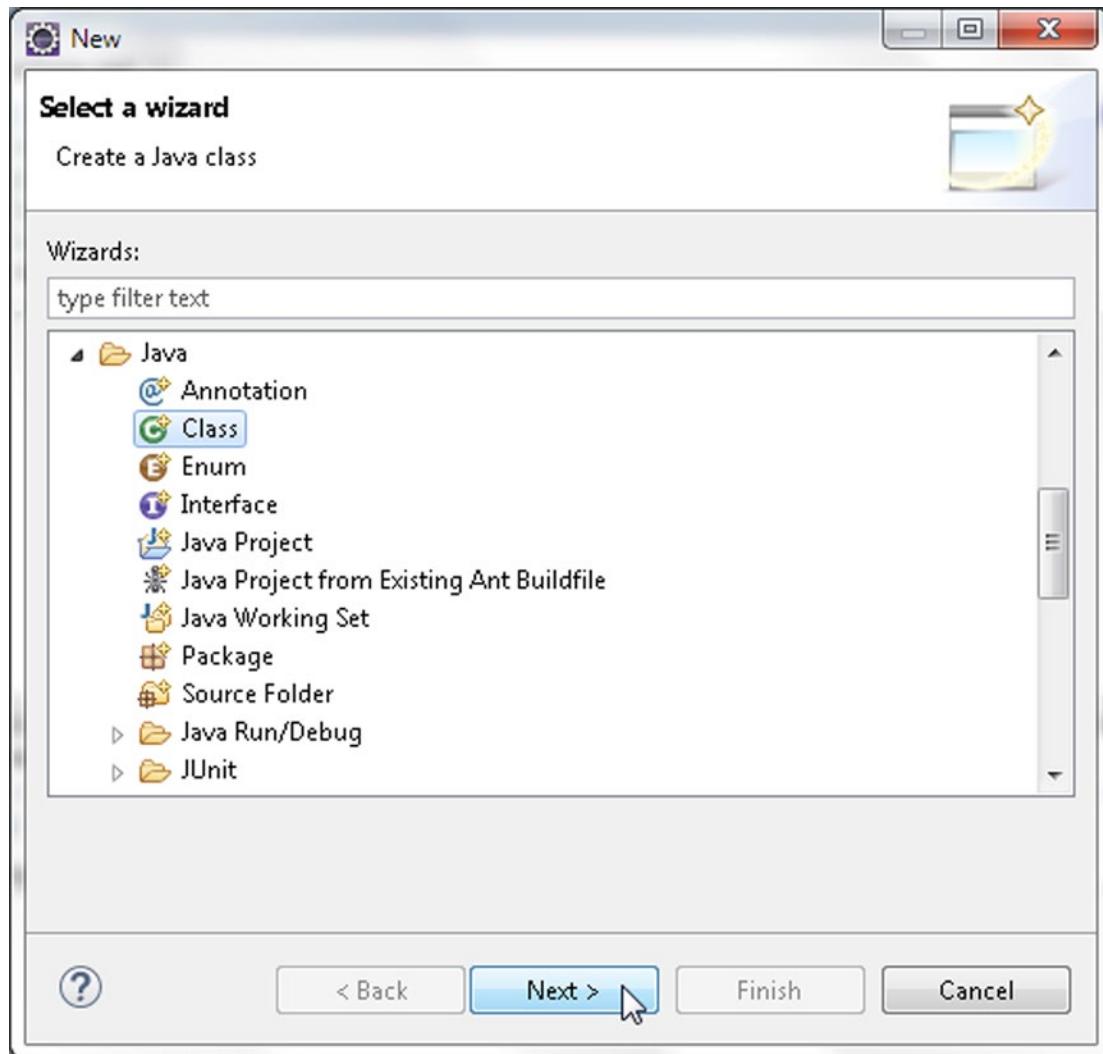


Figure 1-6. Selecting Java ► Class

5. In New Java Class wizard the Source folder is preselected as MongoDBJava/src/main/java. Specify a Package name (mongodb). Specify a class Name (for example, MongoDBClient for the application to connect to MongoDB and get data). Select the public static void main (String[] args) method stub to create the class and click on Finish as shown in Figure 1-7.

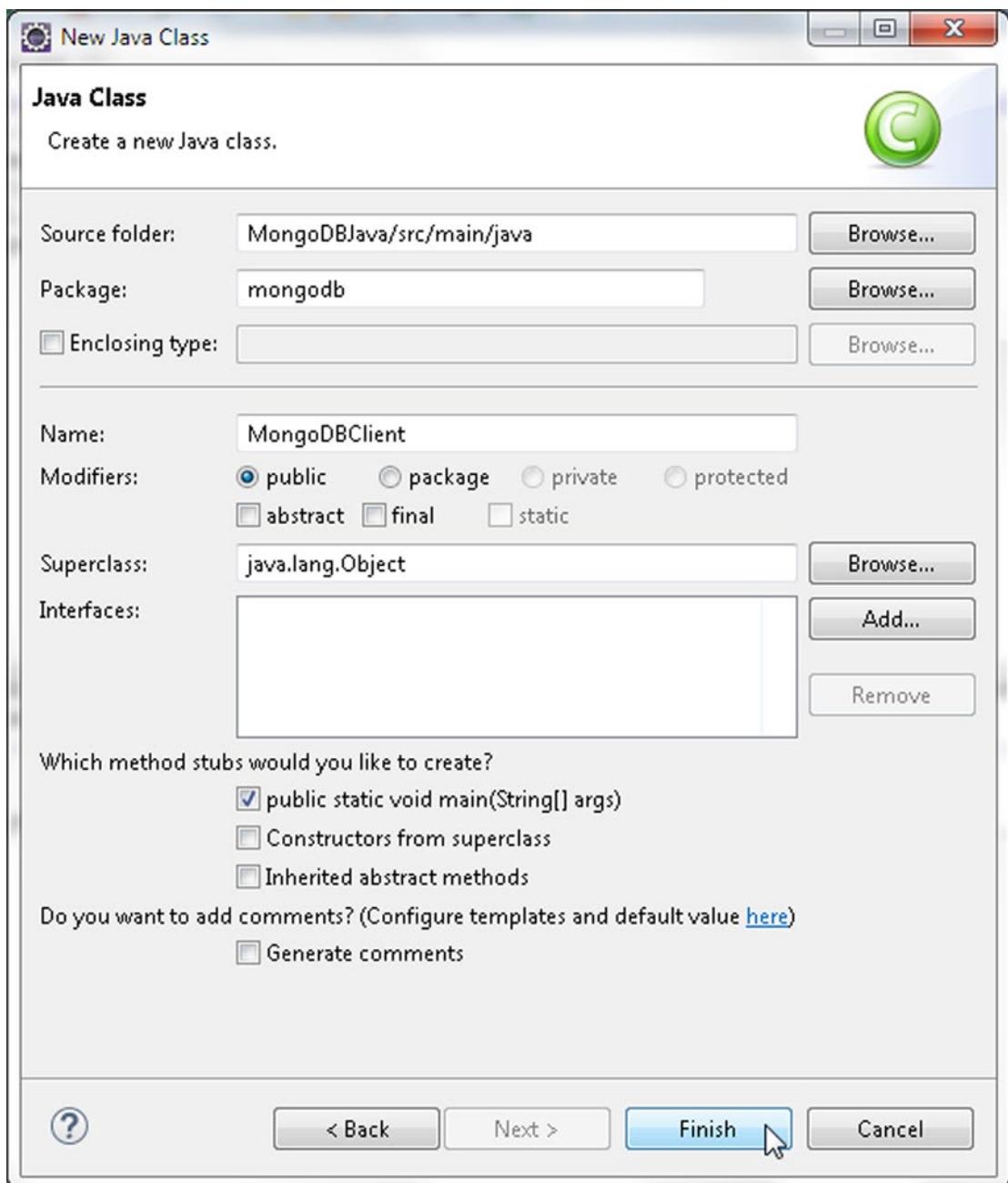


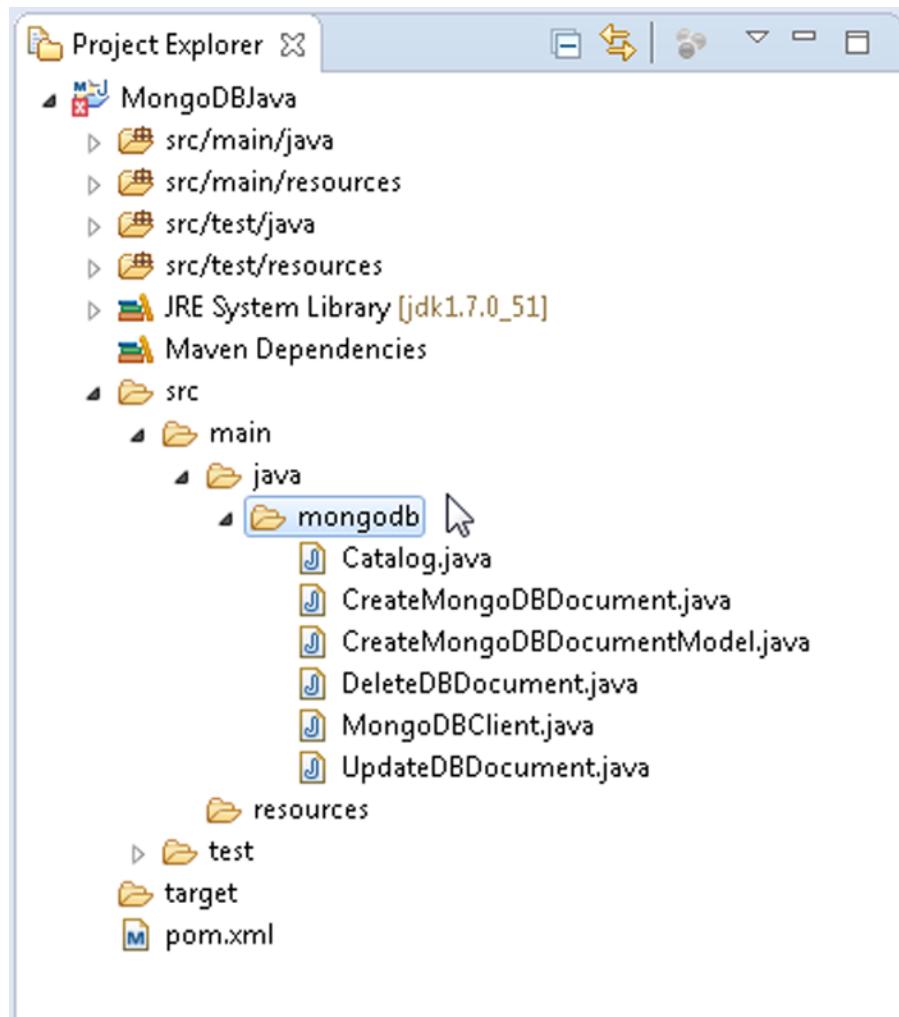
Figure 1-7. Creating a New Class

- The MongoDBClient class gets created in the MongoDBJava project. Similarly add the Java classes listed in Table 1-1 to the same package as the MongoDBClient class: the mongodb package.

Table 1-1. Java Classes

Class	Description
Catalog	The model class to define properties and accessor methods for a record to be added to MongoDB server.
CreateMongoDBDocument	Class to create a MongoDB document.
CreateMongoDBDocumentModel	Class to create a MongoDB document using the model class catalog.
UpdateDBDocument	Class to update a document.
DeleteDBDocument	Class to delete a document.

The Java classes in the Maven project are shown in the Package Explorer in Figure 1-8.

**Figure 1-8.** Java Classes

7. Next, configure the Maven project by adding the required dependencies to the pom.xml configuration file. Add the Java MongoDB Driver dependency to the pom.xml. The pom.xml is listed below. Click on File ► Save All to save the pom.xml.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mongodb.java</groupId>
  <artifactId>MongoDBJava</artifactId>
  <version>1.0.0</version>
  <name>MongoDBJava</name>
  <dependencies>
    <dependency>
      <groupId>org.mongodb</groupId>
      <artifactId>mongo-java-driver</artifactId>
      <version>3.0.3</version>
    </dependency>
  </dependencies>
</project>
```

8. Right-click on the project node in Package Explorer and select Project Properties. In Properties select Java Build Path. Select the Libraries tab and click on Maven Dependencies to expand the node. The MongoDB Java driver should be listed in the Java Build Path as shown in Figure 1-9. Click on OK.

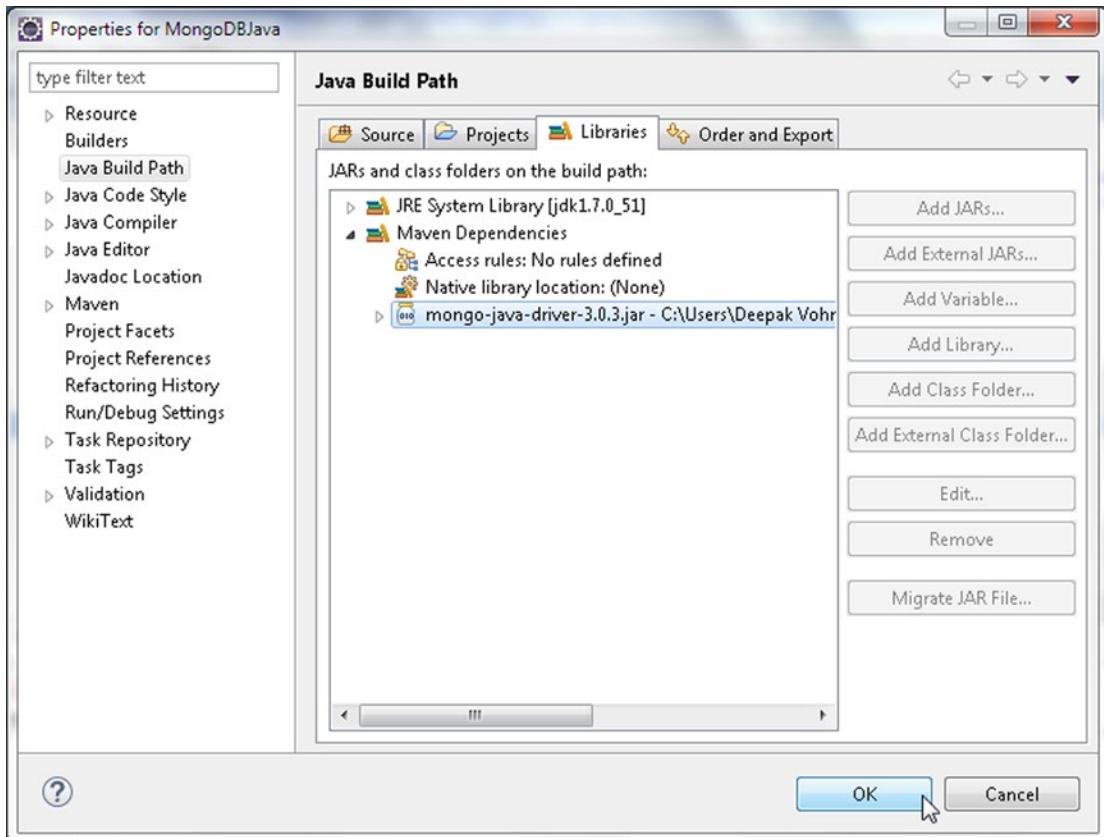


Figure 1-9. Mongo Java Driver in Java Build Path

The Maven project with the Mongo Java driver dependency configured in `pom.xml` is shown in Figure 1-10. The directory structure of the MongoDBJava Maven project is shown in the Package Explorer.

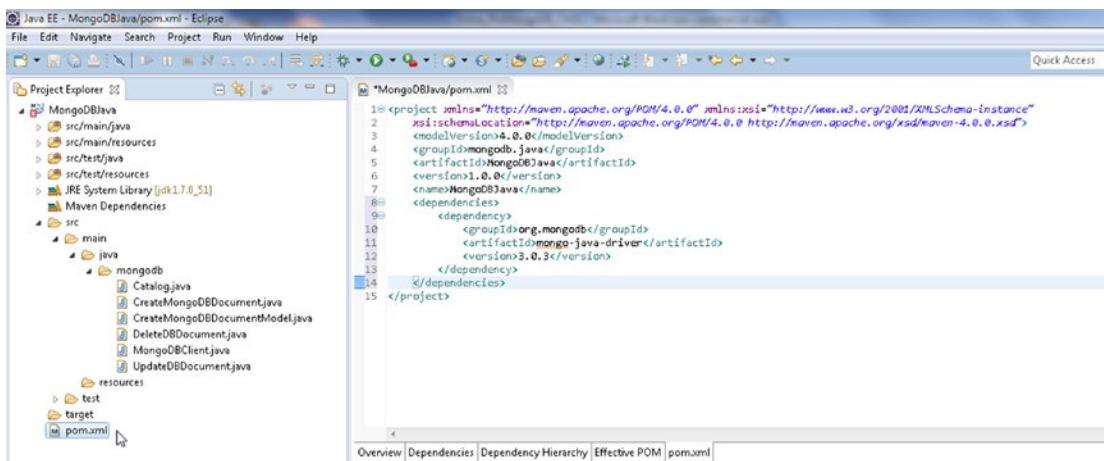


Figure 1-10. Maven Project with Mongo Java Driver Dependency Configured

Creating a BSON Document

In this section we shall add a BSON (Binary JSON) document to MongoDB server. We shall use the `CreateMongoDBDocument.java` application in Eclipse IDE. A MongoDB document is represented with the `org.bson.Document` class. MongoDB stores data in collections. The main packages for MongoDB classes in the MongoDB Java driver are `com.mongodb` and `com.mongodb.client`. A MongoDB client to connect to MongoDB server is represented with the `com.mongodb.MongoClient` class. A `MongoClient` object provides connection pooling, and only one instance is required for the entire instance. The `MongoClient` class provides several constructors, some of which are listed in Table 1-2.

Table 1-2. *MongoClient Class Constructors*

Constructor	Description
<code>MongoClient()</code>	Creates an instance based on a single MongoDB node for localhost and default port 27017.
<code>MongoClient(String host)</code>	Creates an instance based on a single MongoDB node with host specified as a host[:port] String literal.
<code>MongoClient(String host, int port)</code>	Creates an instance based on a single MongoDB node using the specified host and port.
<code>MongoClient(List<ServerAddress> seeds)</code>	Creates an instance using a List of MongoDB servers to select from. The server with the lowest ping time is selected. If the lowest ping time server is down, the next in the list is selected.
<code>MongoClient(List<ServerAddress> seeds, List<MongoCredential> credentialsList)</code>	Same as the previous version except a List of credentials are provided to authenticate connections to the server/s.
<code>MongoClient(List<ServerAddress> seeds, List<MongoCredential> credentialsList, MongoClientOptions options)</code>	Same as the previous version except that Mongo client options are also provided.

1. Create a `MongoClient` instance using the `MongoClient(List<ServerAddress> seeds)` constructor. Supply “localhost” or the IPv4 address of the host and port as 27017.

```
MongoClient mongoClient = new MongoClient  
(Arrays.asList(new ServerAddress("localhost", 27017)));
```

2. When creating many `MongoClient` instances, all resource usage limits apply per `MongoClient` instance. To close an instance you need to call `MongoClient.close()` to clean up resources. A logical database in MongoDB is represented with the `com.mongodb.client.MongoDatabase` interface. Obtain a `com.mongodb.client.MongoDatabase` instance for the “local” database, which is a default MongoDB database instance, using the `getDatabase(String databaseName)` method in `MongoClient` class.

```
MongoDatabase db = mongoClient.getDatabase("local");
```

3. Some of the Mongo client API has been modified in version 3.0.x. For example, a database instance is represented with the `MongoDatabase` in 3.0.x instead of `com.mongodb.DB`. The `getDB(String dbName)` method, which returns a DB instance, in `MongoClient` is deprecated. A database collection in 3.0.x is represented with `com.mongodb.client.MongoCollection<TDocument>` instead of `com.mongodb.DBCollection`. Get all collections from the database instance using the `listCollectionNames()` method in `MongoDatabase`.

```
MongoIterable<String> colls = db.listCollectionNames();
```

4. The `listCollectionNames()` method returns a `MongoIterable<String>` of collections. Iterate over the collection to output the collection names.

```
for (String s : colls) {
    System.out.println(s);
}
```

5. Next, create a new `MongoCollection<Document>` instance using the `getCollection(String collectionName)` method in `MongoDatabase`. Create a collection of `Document` instances called `catalog`. A collection gets created implicitly when the `getCollection(String)` method is invoked.

```
MongoCollection<Document> coll = db.getCollection("catalog");
```

A MongoDB specific BSON object is represented with the `org.bson.Document` class, which implements the `Map` interface among others. The `Document` class provides the following constructors listed in Table 1-3 to create a new instance.

Table 1-3. Document Class Constructors

Constructor	Description
<code>Document()</code>	Creates an empty <code>Document</code> instance.
<code>Document(Map<String, Object> map)</code>	Creates a <code>Document</code> instance initialized with a <code>Map</code> .
<code>Document(String key, Object value)</code>	Creates a <code>Document</code> instance initialized with a key/value pair.

The `Document` class provides some other utility methods, some of which are in Table 1-4.

Table 1-4. Document Class Utility Methods

Method	Description
<code>append(String key, Object value)</code>	Appends a key/value pair to a <code>Document</code> object and returns a new instance.
<code>toString()</code>	Returns a String representation of the object.

6. Create a Document instance using the Document(String key, Object value) constructor and use the append(String key, Object value) method to append key/value pairs. The append() method may be invoked multiple times in sequence to add multiple key/value pairs. Add key/value pairs for the journal, publisher, edition, title, and author fields.

```
Document catalog = new Document("journal", "Oracle Magazine")
.append("publisher", "Oracle Publishing")
.append("edition", "November December 2013")
.append("title", "Engineering as a Service").append("author",
"David A. Kelly");
```

7. The MongoCollection<TDocument> interface provides insertOne(TDocument document) method to add a document(s) to a collection. Add the catalog Document to the MongoCollection<TDocument> instance for the catalog collection.

```
coll.insertOne(catalog);
```

8. The MongoCollection<TDocument> interface provides overloaded find() method to find a Document instance. Next, obtain the document added using the find() method. Furthermore, the find() method returns an iterable collection from which we obtain the first document using the first() method.

```
Document dbObj = coll.find().first();
```

9. Output the Document object found as such and also by iterating over the Set<E> obtained from the Document using the keySet() method. The keySet() method returns a Set<String>. Create an Iterator from the Set<String> using the iterator() method. While the Iterator has elements as determined by the hasNext() method, obtain the elements using the next() method. Each element is a key in the Document fetched. Obtain the value for the key using the get(String key) method in Document.

```
System.out.println(dbObj);
Set<String> set = dbObj.keySet();
Iterator iter = set.iterator();
while(iter.hasNext()){
Object obj= iter.next();
System.out.println(obj);
System.out.println(dbObj.get(obj.toString()));
}
```

10. Close the MongoClient instance.

```
mongoClient.close();
```

The CreateMongoDBDocument class is listed below.

```
package mongodb;

import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.MongoIterable;

public class CreateMongoDBDocument {

    public static void main(String[] args) {

        MongoClient mongoClient = new MongoClient(
            Arrays.asList(new ServerAddress("localhost", 27017)));
        for (String s : mongoClient.listDatabaseNames()) {
            System.out.println(s);
        }
        MongoDatabase db = mongoClient.getDatabase("local");
        MongoIterable<String> colls = db.listCollectionNames();
        System.out.println("MongoDB Collection Names: ");
        for (String s : colls) {
            System.out.println(s);
        }
        MongoCollection<Document> coll = db.getCollection("catalog");
        Document catalog = new Document("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Engineering as a Service")
            .append("author", "David A. Kelly");
        coll.insertOne(catalog);
        Document dbObj = coll.find().first();
        System.out.println(dbObj);
        Set<String> set = catalog.keySet();
        Iterator<String> iter = set.iterator();
        while (iter.hasNext()) {
            Object obj = iter.next();
            System.out.println(obj);
            System.out.println(dbObj.get(obj));
        }
        mongoClient.close();
    }
}
```

11. To run the `CreateMongoDBDocument` application, right-click on the `CreateMongoDBDocument.java` file in Package Explorer and select Run As ► Java Application as shown in Figure 1-11.

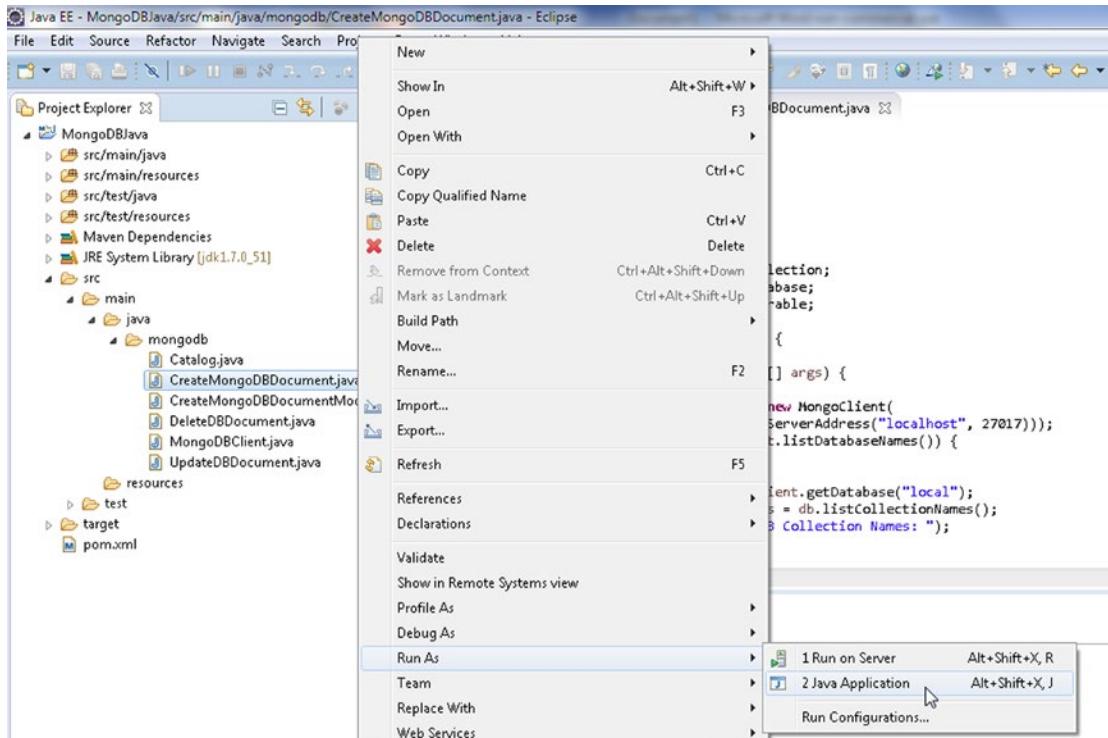
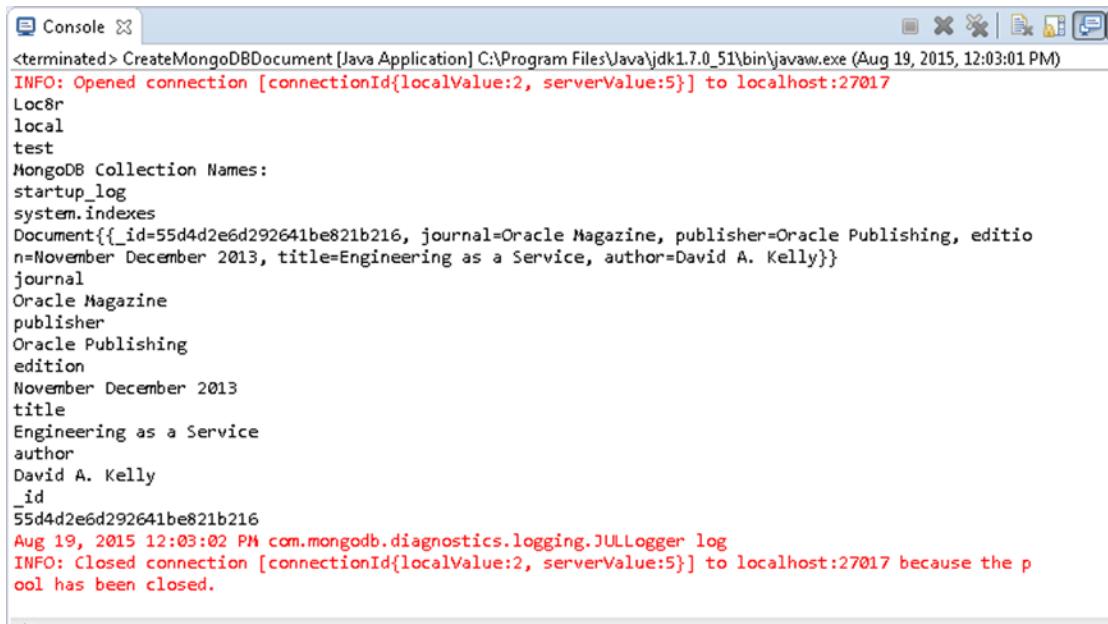


Figure 1-11. Running `CreateMongoDBDocument.java` Application

A new BSON document gets stored in a new collection catalog in MongoDB database. The document stored is also output as such and as key/value pairs as shown in Figure 1-12.



The screenshot shows a Java application window titled "Console". The console output is as follows:

```

<terminated> CreateMongoDBDocument [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 19, 2015, 12:03:01 PM)
INFO: Opened connection [connectionId{localValue:2, serverValue:5}] to localhost:27017
Loc8r
local
test
MongoDB Collection Names:
startup_log
system.indexes
Document{{_id=55d4d2e6d292641be821b216, journal=Oracle Magazine, publisher=Oracle Publishing, edition=November December 2013, title=Engineering as a Service, author=David A. Kelly}}
journal
Oracle Magazine
publisher
Oracle Publishing
edition
November December 2013
title
Engineering as a Service
author
David A. Kelly
_id
55d4d2e6d292641be821b216
Aug 19, 2015 12:03:02 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Closed connection [connectionId{localValue:2, serverValue:5}] to localhost:27017 because the pool has been closed.

```

Figure 1-12. Outputting Document Stored in MongoDB

Using a Model to Create a BSON Document

In the previous section we constructed the documents to add to a collection using the `append(String key, Object value)` method in `Document` class and added the documents to a collection using the `insertOne(TDocument document)` method in `MongoCollection<TDocument>` interface. A `Document` object may also be constructed using a model class that represents the objects in a collection. The `MongoCollection<TDocument>` interface provides the overloaded `insertMany()` method to add a list of documents as discussed in Table 1-5.

Table 1-5. Overloaded `insertMany()` Method

Method	Description
<code>insertMany(List<? extends TDocument> documents)</code>	Inserts one or more documents.
<code>insertMany(List<? extends TDocument> documents, InsertManyOptions options)</code>	Inserts one or more documents using the specified insert options. The only option supported is to insert the documents in the order provided.

In this section we shall construct a list of documents using a model class. Subsequently, we shall insert the list into a collection using one of the `insertMany()` methods.

1. First, create a model class `Catalog` with the following fields:

- `catalogId`
- `journal`
- `publisher`
- `edition`
- `title`
- `author`

2. The model class extends the `BasicDBObject` class, which is a basic implementation of a MongoDB specific BSON object, and implements the `Serializable` interface. The class implements the `Serializable` interface to serialize a model class object to a cache when persisted to a database. To associate a version number with a serializable class by serialization runtime, specify a `serialVersionUID` variable with scope `private`.

```
private static final long serialVersionUID = 1L;
```

3. Specify a class constructor that takes all the fields as args. The model class `Catalog` is listed below.

```
package mongodb;
import java.io.Serializable;
import com.mongodb.BasicDBObject;
public class Catalog extends BasicDBObject implements Serializable {

    private static final long serialVersionUID = 1L;

    private String catalogId;
    private String journal;
    private String publisher;
    private String edition;
    private String title;
    private String author;    public Catalog() {
        super();
    }

    public Catalog(String catalogId, String journal, String publisher,
                  String edition, String title, String author) {
        this.catalogId = catalogId;
        this.journal = journal;
        this.publisher = publisher;
        this.edition = edition;
        this.title = title;
        this.author = author;
    }
}
```

Next, we shall use the model class `Catalog` to construct documents and add the documents to MongoDB database. We shall use the `CreateMongoDBDocumentModel` class to construct and add documents to MongoDB.

1. First, create a `MongoClient` instance as discussed previously.

```
MongoClient mongoClient = new MongoClient
(NSArray.asList(new ServerAddress("localhost", 27017)));
```

2. Also create a `MongoDatabase` instance for the local database using the `getDatabase(String databaseName)` method in `MongoClient` class.

```
MongoDatabase db = mongoClient.getDatabase("local");
```

The `MongoDatabase` interface provides the overloaded methods listed in Table 1-6 for getting or creating a collection represented with the `MongoCollection<TDocument>` interface.

Table 1-6. Overloaded `getCollection()` Method

Method	Description
<code>getCollection(String collectionName)</code>	Gets a collection as a <code>MongoCollection<Document></code> instance. If the collection does not already exist, it creates the collection.
<code>getCollection(String collectionName, Class<TDocument> documentClass)</code>	Gets a collection as a <code>MongoCollection<TDocument></code> instance. If the collection does not already exist, it creates the collection. The second argument represents the document class. The only difference between <code>MongoCollection<Document></code> and <code>MongoCollection<TDocument></code> is the type parameter; <code>TDocument</code> represents the type of the document and <code>Document</code> the document.

3. Create a collection from the `MongoDatabase` instance created earlier using the `getCollection(String collectionName)` method.

```
MongoCollection<Document> coll = db.getCollection("catalog");
```

4. As the `Catalog` class extends the `BasicDBObject` class it also represents a document object that may be stored in MongoDB database. Create instances of `Catalog` and set the object fields using the `put(String key, Object value)` method in `BasicDBObject` class.

```
Catalog catalog1 = new Catalog();
catalog1.put("catalogId", "catalog1");
catalog1.put("journal", "Oracle Magazine");
catalog1.put("publisher", "Oracle Publishing");
catalog1.put("edition", "November December 2013");
catalog1.put("title", "Engineering as a Service");
catalog1.put("author", "David A. Kelly");
```

```
Catalog catalog2 = new Catalog();
catalog2.put("catalogId", "catalog2");
catalog2.put("journal", "Oracle Magazine");
catalog2.put("publisher", "Oracle Publishing");
catalog2.put("edition", "November December 2013");
catalog2.put("title", "Quintessential and Collaborative");
catalog2.put("author", "Tom Haunert");
```

5. Create a Document instance and add the Catalog objects to the Document as key/value pairs using the append(String key, Object value) method.

```
Document documentSet = new Document();
documentSet.append("catalog1", catalog1);
documentSet.append("catalog2", catalog2);
```

6. As the argument to the insertMany() method for adding documents is required to be of type List<E> we need to create a List<E> instance using a ArrayList<E> constructor.

```
ArrayList<Document> arrayList = new ArrayList<Document>();
```

7. Add the Document instance with key/value pairs added to it to the ArrayList<E> using the add(E e) method.

```
arrayList.add(documentSet);
```

8. Add the ArrayList<E> instance to the MongoCollection<TDocument> instance using the insertMany(List<? extends TDocument> documents) method.

```
coll.insertMany(arrayList);
```

9. To verify that the documents have been added get the documents using the find() method in MongoCollection<TDocument>. The find() method returns as the result a FindIterable<TDocument> object.

```
FindIterable<Document> iterable = coll.find();
```

Subsequently, output the key/value pairs stored in the FindIterable<TDocument> object. Using an enhanced for loop obtain the Document instances in the FindIterable<TDocument> and obtain the key set associated with each Document instance using the keySet() method, which returns a Set<String> object. Create an iterator represented with an Iterator<String> object using the iterator() method in Set<E>. Using a while loop iterate over the key set to output the document key for each Document and the associated Document object.

```
FindIterable<Document> iterable = coll.find();
String documentKey = null;
for (Document document : iterable) {
    Set<String> keySet = document.keySet();
    Iterator<String> iter = keySet.iterator();
    while (iter.hasNext()) {
```

```

        documentKey = iter.next();
        System.out.println(documentKey);
        System.out.println(document.get(documentKey));
    }
}

```

10. Close the MongoClient object using the `close()` method.

```
mongoClient.close();
```

The `CreateMongoDBDocumentModel` class is listed below.

```

package mongodb;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

public class CreateMongoDBDocumentModel {
    public static void main(String[] args) {
        MongoClient mongoClient = new MongoClient(
            Arrays.asList(new ServerAddress("localhost", 27017)));

        MongoDatabase db = mongoClient.getDatabase("local");

        MongoCollection<Document> coll = db.getCollection("catalog");

        Catalog catalog1 = new Catalog();
        catalog1.put("catalogId", "catalog1");
        catalog1.put("journal", "Oracle Magazine");
        catalog1.put("publisher", "Oracle Publishing");
        catalog1.put("edition", "November December 2013");
        catalog1.put("title", "Engineering as a Service");
        catalog1.put("author", "David A. Kelly");

        Catalog catalog2 = new Catalog();
        catalog2.put("catalogId", "catalog2");
        catalog2.put("journal", "Oracle Magazine");
        catalog2.put("publisher", "Oracle Publishing");
        catalog2.put("edition", "November December 2013");
        catalog2.put("title", "Quintessential and Collaborative");
        catalog2.put("author", "Tom Haunert");
    }
}

```

```

Document documentSet = new Document();
documentSet.append("catalog1", catalog1);
documentSet.append("catalog2", catalog2);
ArrayList<Document> arrayList = new ArrayList<Document>();
arrayList.add(documentSet);
coll.insertMany(arrayList);
FindIterable<Document> iterable = coll.find();
String documentKey = null;
for (Document document : iterable) {
    Set<String> keySet = document.keySet();
    Iterator<String> iter = keySet.iterator();
    while (iter.hasNext()) {
        documentKey = iter.next();
        System.out.println(documentKey);
        System.out.println(document.get(documentKey));
    }
}
mongoClient.close();
}
}

```

11. Next, run the `CreateMongoDBDocumentModel` application. Before running the application drop the `catalog` collection using `db.catalog.drop()` in mongo shell as we shall be creating an empty `catalog` collection in the application to add documents. Right-click on the `CreateMongoDBDocumentModel.java` file in Package Explorer and select Run As ► Java Application as shown in Figure 1-13.

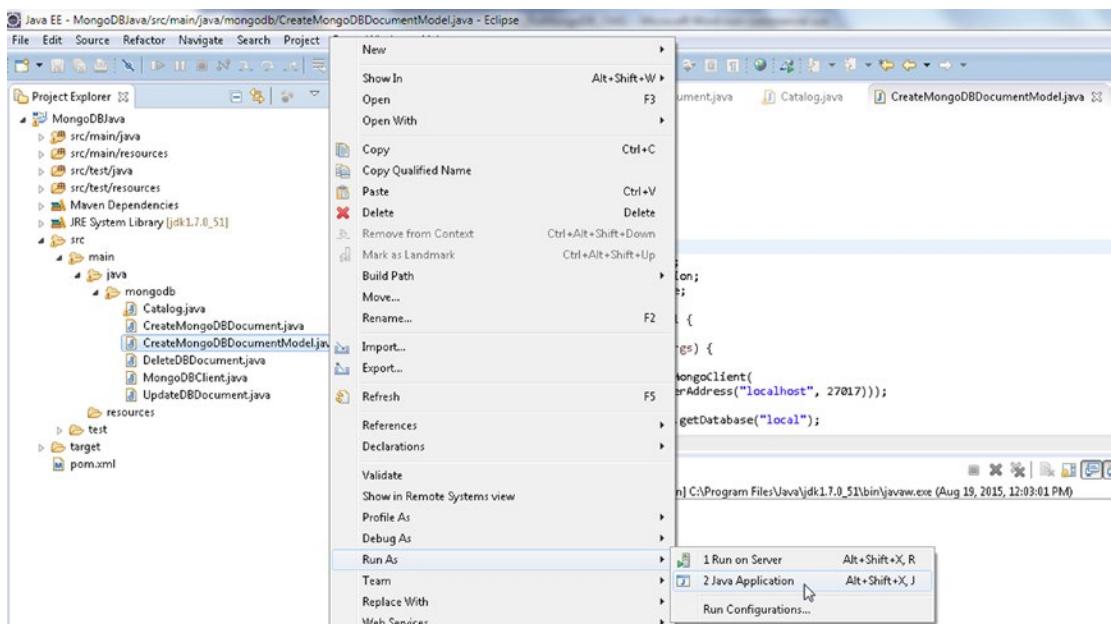
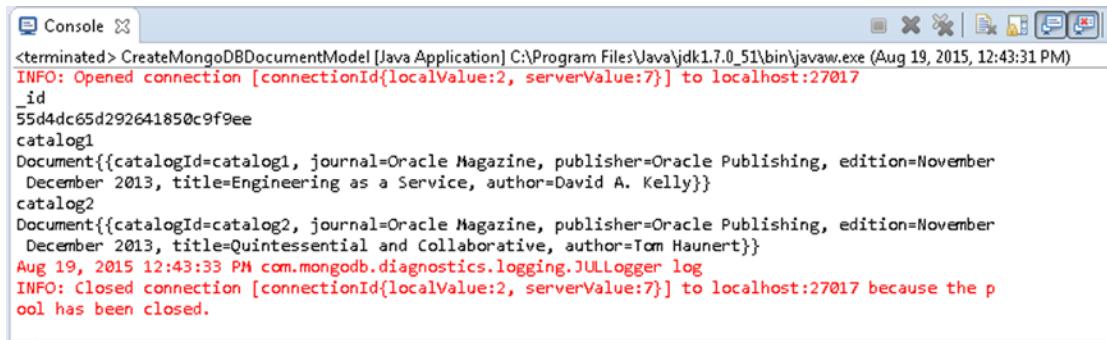


Figure 1-13. Running the `CreateMongoDBDocumentModel.java` Application

One document gets added to the MongoDB collection catalog as indicated by the one `_id` fetched. The document added has two key/value pairs. Subsequently, the key/value pairs for the Catalog instances added get output as shown in Figure 1-14. The `_id` would most likely be different than what is shown in Figure 1-14 as it is generated automatically.



The screenshot shows a Java application console window titled "Console". The output from the application is displayed, showing the creation of a MongoDB document model. The log includes:

- `<terminated> CreateMongoDBDocumentModel [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 19, 2015, 12:43:31 PM)`
- `INFO: Opened connection [connectionId{localValue:2, serverValue:7}] to localhost:27017`
- `_id`
- `55d4dc65d292641850c9f9ee`
- `catalog1`
- `Document{{catalogId=catalog1, journal=Oracle Magazine, publisher=Oracle Publishing, edition=November December 2013, title=Engineering as a Service, author=David A. Kelly}}`
- `catalog2`
- `Document{{catalogId=catalog2, journal=Oracle Magazine, publisher=Oracle Publishing, edition=November December 2013, title=Quintessential and Collaborative, author=Tom Haunert}}`
- `Aug 19, 2015 12:43:33 PM com.mongodb.diagnostics.logging.JULLogger log`
- `INFO: Closed connection [connectionId{localValue:2, serverValue:7}] to localhost:27017 because the pool has been closed.`

Figure 1-14. Output from the `CreateMongoDBDocumentModel.java` Application

Getting Data from MongoDB

In this section we shall fetch data from MongoDB. We shall use the `MongoDBClient` application in this section. The `MongoCollection<TDocument>` interface provides the overloaded methods discussed in Table 1-7 to find documents.

Table 1-7. Overloaded `find()` Methods

Method	Description
<code>find()</code>	Finds all the documents in the collection and returns a <code>FindIterable<TDocument></code> instance.
<code>find(Bson filter)</code>	Finds all the documents in the collection using the specified query filter and returns a <code>FindIterable<TDocument></code> instance.
<code>find(Bson filter, Class<TResult> resultClass)</code>	Finds all the documents in the collection using the specified query filter and result class and returns a <code><TResult> FindIterable<TResult></code> instance.
<code>find(Class<TResult> resultClass)</code>	Finds all the documents in the collection using the specified result class and returns a <code><TResult> FindIterable<TResult></code> instance.

1. Create a `MongoClient` instance, a `MongoDatabase` instance, and a `MongoCollection<TDocument>` instance as discussed earlier.

```

MongoClient mongoClient = new MongoClient(Arrays.asList(new
ServerAddress("localhost", 27017)));
MongoDatabase db = mongoClient.getDatabase("local");
MongoCollection<Document> coll = db.getCollection("catalog");

```

2. Create two Catalog instances and add the Catalog instances to the MongoCollection<TDocument> instance using the insertOne(TDocument document) method.

```
Document catalog = new Document("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013")
    .append("title", "Engineering as a Service")
    .append("author", "David A. Kelly");
coll.insertOne(catalog);

catalog = new Document("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013")
    .append("title", "Quintessential and Collaborative")
    .append("author", "Tom Haunert");
coll.insertOne(catalog);
```

3. Subsequently, find the documents added using the find() method, which returns the result as a FindIterable <TDocument>.

```
FindIterable<Document> iterable = coll.find();
```

4. Using an enhanced for loop iterate over the FindIterable<TDocument> to obtain the Document instances stored and obtain the key set associated with each Document instance. Create a iterator over the key set and using a while loop iterate over the key set to output each document key and Document object associated with each document key.

```
String documentKey = null;
for (Document document : iterable) {
    Set<String> keySet = document.keySet();
    Iterator<String> iter = keySet.iterator();
    while (iter.hasNext()) {
        documentKey = iter.next();
        System.out.println(documentKey);
        System.out.println(document.get(documentKey));
    }
}
```

The MongoDBClient class is listed below.

```
package mongodb;

import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.FindIterable;
```

```

import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
public class MongoDBClient {
    public static void main(String[] args) {
        MongoClient mongoClient = new MongoClient(
            Arrays.asList(new ServerAddress("localhost", 27017)));
        MongoDatabase db = mongoClient.getDatabase("local");
        MongoCollection<Document> coll = db.getCollection("catalog");
        Document catalog = new Document("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Engineering as a Service")
            .append("author", "David A. Kelly");
        coll.insertOne(catalog);

        catalog = new Document("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Quintessential and Collaborative")
            .append("author", "Tom Haunert");
        coll.insertOne(catalog);
        FindIterable<Document> iterable = coll.find();
        String documentKey = null;
        for (Document document : iterable) {
            Set<String> keySet = document.keySet();
            Iterator<String> iter = keySet.iterator();
            while (iter.hasNext()) {
                documentKey = iter.next();
                System.out.println(documentKey);
                System.out.println(document.get(documentKey));
            }
        }
        mongoClient.close();
    }
}

```

5. Before running the MongoDBClient application drop catalog collection from local database using the following commands from Mongo shell (which is discussed in more detail in the next chapter). The Mongo shell is started using the `mongo` command. Start the Mongo shell in a new command window and the MongoDB server instance should be running when the mongo shell commands are run.

```

>mongo
>use local
>db.catalog.drop()

```

6. To run the MongoDBClient application right-click on the `MongoDBClient.java` file in Package Explorer and select Run As ➤ Java Application as shown in Figure 1-15.

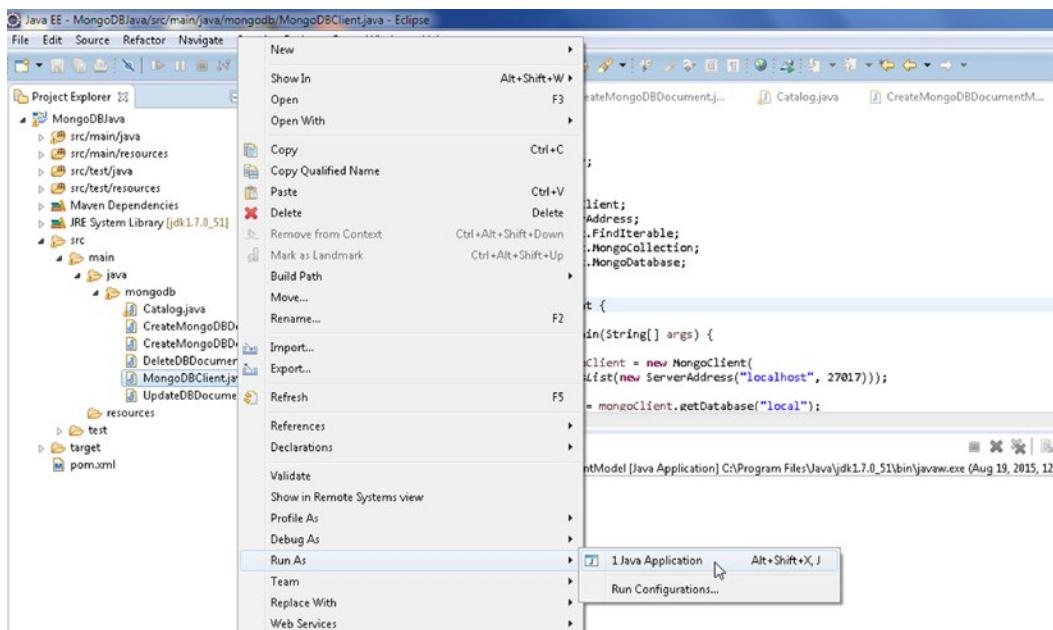


Figure 1-15. Running the MongoClient.java Application

The output from the application is displayed in the Eclipse Console as shown in Figure 1-16. The two documents added get fetched and the associated key/value pairs get output.

```

<terminated> MongoDBClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 19, 2015, 1:02:05 PM)
INFO: Opened connection [connectionId{localValue:2, serverValue:9}] to localhost:27017
_id
55d4e0bed292641250e3efdc
journal
Oracle Magazine
publisher
Oracle Publishing
edition
November December 2013
title
Engineering as a Service
author
David A. Kelly
_id
55d4e0bfd292641250e3efdd
journal
Oracle Magazine
publisher
Oracle Publishing
edition
November December 2013
title
Quintessential and Collaborative
author
Tom Haunert
Aug 19, 2015 1:02:07 PM com.mongodb.diagnostics.logging.JULLoader log
INFO: Closed connection [connectionId{localValue:2, serverValue:9}] to localhost:27017 because the pool has been closed.

```

Figure 1-16. Finding Documents and Outputting Key/Value Pairs

Updating Data in MongoDB

In this section we shall update MongoDB data. We shall be using the `UpdateDBDocument` application. The `MongoCollection<TDocument>` class provides several methods, some of them overloaded, to find and update data as discussed in Table 1-8.

Table 1-8. *MongoCollection<TDocument> Methods to Update or Replace Documents*

Method	Description
<code>findOneAndReplace(Bson filter, TDocument replacement)</code>	Finds a document using the specified query filter and replaces it with the replacement document.
<code>findOneAndReplace(Bson filter, TDocument replacement, FindOneAndReplaceOptions options)</code>	Finds a document using the specified query filter and <code>FindOneAndReplaceOptions</code> options and replaces it with the replacement document. The <code>FindOneAndReplaceOptions</code> options include the maximum time to find and replace, whether to return the original document or the replacement document, the sort criteria to apply, whether to upsert the document if the query filter does not find a document, and which document fields to return. Upsert is the term used to insert a document if the document to be replaced or updated is not found.
<code>findOneAndUpdate(Bson filter, Bson update)</code>	Finds a document using the specified query filter and replaces it with the update document. The update must include only update operators.
<code>findOneAndUpdate(Bson filter, Bson update, FindOneAndUpdateOptions options)</code>	Finds a document using the specified query filter and replaces it with the update document. The update must include only update operators. The <code>FindOneAndUpdateOptions</code> options are also provided. The <code>FindOneAndUpdateOptions</code> options include the maximum time to find and update, whether to return the original document or the updated document, the sort criteria to apply, whether to upsert the document if the query filter does not find a document, and which document fields to return.
<code>replaceOne(Bson filter, TDocument replacement)</code>	Replaces a document using the specified query filter and the replacement document.
<code>replaceOne(Bson filter, TDocument replacement, UpdateOptions updateOptions)</code>	Replaces a document using the specified query filter and the replacement document. The <code>UpdateOptions</code> options are also provided. The <code>UpdateOptions</code> options include whether to upsert the document if the query filter does not find a document.
<code>updateMany(Bson filter, Bson update)</code>	Updates one or more documents using a query filter and an update document.

(continued)

Table 1-8. (continued)

Method	Description
updateMany(Bson filter, Bson update, UpdateOptions updateOptions)	Updates one or more documents using a query filter and an update document. The UpdateOptions are also provided. The UpdateOptions options include whether to upsert the document if the query filter does not find a document.
updateOne(Bson filter, Bson update)	Updates one document using a query filter and an update document.
updateOne(Bson filter, Bson update, UpdateOptions updateOptions)	Updates one document using a query filter and an update document. The UpdateOptions are also provided. The UpdateOptions options include whether to upsert the document if the query filter does not find a document.

Some of the methods listed support only update operators in the update document. The update operators that may be applied on document fields are discussed in Table 1-9.

Table 1-9. Update Operators

Method	Description
\$inc	Increments the value of a field. The increment must be a positive or negative value.
\$mul	Multiplies the value of a field.
\$rename	Renames a field.
\$setOnInsert	Sets the value of a field if the update results in an insert.
\$set	Sets the value of a field in a document.
\$unset	Unsets the value of a field in a document.
\$min	Updates a field only if the specified value is less than the present field value.
\$max	Updates a field only if the specified value is more than the present field value.
\$currentDate	Sets the value of the field to the current date as a Date or Timestamp.

1. In the UpdateDBDocument application create a MongoClient instance as discussed earlier and create a MongoDB instance for the local database. Subsequently, create a MongoCollection<TDocument> instance for the catalog collection. Add two instances of Catalog objects to the catalog collection using the insertOne(TDocument document) method.

```
Document catalog = new Document("catalogId", "catalog1")
.append("journal", "Oracle Magazine")
.append("publisher", "Oracle Publishing")
.append("edition", "November December 2013")
.append("title", "Engineering as a Service")
.append("author", "David A. Kelly");
coll.insertOne(catalog);
```

```

catalog = new Document("catalogId", "catalog2")
.append("journal", "Oracle Magazine")
.append("publisher", "Oracle Publishing")
.append("edition", "November December 2013")
.append("title", "Quintessential and Collaborative")
.append("author", "Tom Haunert");
coll.insertOne(catalog);

```

2. As an example of using the updateOne(Bson filter, Bson update) method, update the edition and author fields of the Document instance with catalogId catalog1 using the update operator \$set.

```

coll.updateOne(new Document("catalogId", "catalog1"), new Document("$set", new
Document("edition", "11-12 2013").append("author", "Kelly, David A.")));

```

3. As an example of using the updateMany(Bson filter, Bson update) method, update the journal field of all Document instances using update operator \$set.

```

coll.updateMany(new Document("journal", "Oracle Magazine"),
new Document("$set", new Document("journal", "OracleMagazine")));

```

4. As an example of using the replaceOne(Bson filter, TDocument replacement, UpdateOptions updateOptions) method, replace the Document instance with catalogId catalog3, which does not exist, with a new Document instance. Provide an UpdateOptions argument to upsert the Document if the query filter does not return a Document instance.

```

UpdateResult result = coll.replaceOne(new Document("catalogId",
"catalog3"), new Document("catalogId", "catalog3").append("journal",
"Oracle Magazine").append("publisher", "Oracle Publishing").
append("edition", "November December 2013").append("title",
"Engineering as a Service").append("author", "David A. Kelly"),
new UpdateOptions().upsert(true));

```

5. The replaceOne() method returns an UpdateResult object. Output the following:

- The number of documents matched using the getMatchedCount() method of UpdateResult.
- The number of documents modified using the getModifiedCount() method. Not all matched documents may be modified.
- The _id field value for the upserted document using the getUpsertedId() method. The _id field value has to be obtained using the asObjectId() method invocation followed by the getValue() method invocation.

```

System.out.println("Number of documents matched: " + result.getMatchedCount());
System.out.println("Number of documents modified: " + result.getModifiedCount());
System.out.println("Upserted Document Id: " + result.getUpsertedId().asObjectId().
getValue());

```

6. To verify that the documents got updated or replaced, output all the documents in the catalog collection. Create a `FindIterable<TDocument>` for the documents in the catalog collection using the `find()` method. Subsequently, use an enhanced for loop to obtain the `Document` instances and output the key/value pairs in each `Document` instance as discussed earlier.

The `UpdateDBDocument` application is listed below.

```
package mongodb;

import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.UpdateOptions;
import com.mongodb.client.result.UpdateResult;

public class UpdateDBDocument {

    public static void main(String[] args) {
        MongoClient mongoClient = new MongoClient(
            Arrays.asList(new ServerAddress("localhost", 27017)));
        MongoDatabase db = mongoClient.getDatabase("local");
        MongoCollection<Document> coll = db.getCollection("catalog");
        Document catalog = new Document("catalogId", "catalog1")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Engineering as a Service")
            .append("author", "David A. Kelly");
        coll.insertOne(catalog);
        catalog = new Document("catalogId", "catalog2")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Quintessential and Collaborative")
            .append("author", "Tom Haunert");
        coll.insertOne(catalog);
        coll.updateOne(
            new Document("catalogId", "catalog1"),
            new Document("$set", new Document("edition", "11-12 2013")
                .append("author", "Kelly, David A.")));
        coll.updateMany(new Document("journal", "Oracle Magazine"),
            new Document("$set", new Document("journal", "OracleMagazine")));
    }
}
```

```

UpdateResult result = coll.replaceOne(
    new Document("catalogId", "catalog3"),
    new Document("catalogId", "catalog3")
        .append("journal", "Oracle Magazine")
        .append("publisher", "Oracle Publishing")
        .append("edition", "November December 2013")
        .append("title", "Engineering as a Service")
        .append("author", "David A. Kelly"),
    new UpdateOptions().upsert(true));
System.out.println("Number of documents matched: "
    + result.getMatchedCount());

System.out.println("Number of documents modified: "
    + result.getModifiedCount());
System.out.println("Upserted Document Id: "
    + result.getUpsertedId().asObjectId().getValue());
FindIterable<Document> iterable = coll.find();
String documentKey = null;
for (Document document : iterable) {
    Set<String> keySet = document.keySet();
    Iterator<String> iter = keySet.iterator();
    while (iter.hasNext()) {
        documentKey = iter.next();
        System.out.println(documentKey);
        System.out.println(document.get(documentKey));
    }
}
mongoClient.close();
}

```

7. Again, before running the application drop the catalog collection with `db.catalog.drop()` command in mongo shell. To run the `UpdateDBDocument.java` application right-click on `UpdateDBDocument.java` in Package Explorer and select Run As ➤ Java Application as shown in Figure 1-17.

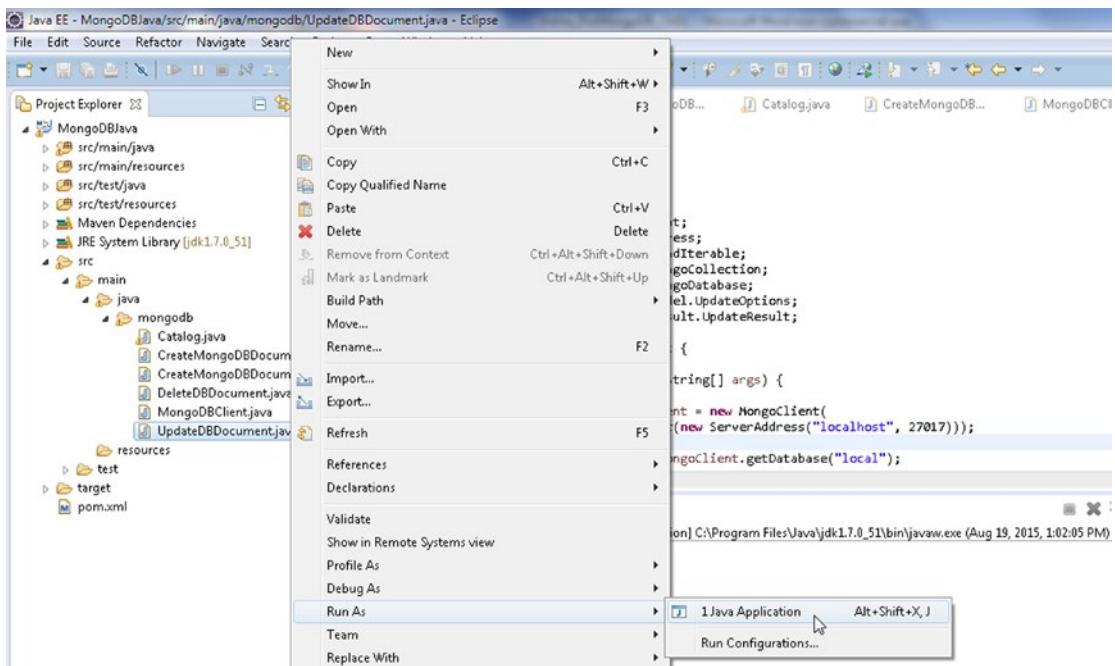


Figure 1-17. Running the `UpdateDBDocument.java` Application

The output from the `UpdateDBDocument.java` application is as follows.

```
Number of documents matched: 0
Number of documents modified: 0
Upserted Document Id: 55d4e56c0bc271d4a7002749
_id
55d4e56cd292641a003cb55a
catalogId
catalog1
journal
OracleMagazine
publisher
Oracle Publishing
edition
11-12 2013
title
Engineering as a Service
author
Kelly, David A.
```

```
_id  
55d4e56cd292641a003cb55b  
catalogId  
catalog2  
journal  
OracleMagazine  
publisher  
Oracle Publishing  
edition  
November December 2013  
title  
Quintessential and Collaborative  
author  
Tom Haunert  
_id  
55d4e56c0bc271d4a7002749  
catalogId  
catalog3  
journal  
Oracle Magazine  
publisher  
Oracle Publishing  
edition  
November December 2013  
title  
Engineering as a Service  
author  
David A. Kelly
```

The `updateOne()` method example updates the `edition` and `author` fields of the document with `catalogId` as `catalog1` using the update operator `$set`. The `updateMany()` method example sets the `journal` field of all documents to `OracleMagazine` using the update operator `$set`. As indicated in the output, the number of documents matched and modified are both 0 for the `replaceOne()` method example. Because `UpdateOptions` are set to upsert a document, a new document gets added when a `Document` instance for `catalogId catalog3` is not found.

Deleting Data in MongoDB

In this section we shall delete documents using the `DeleteDBDocument` application. The `MongoCollection<TDocument>` interface provides several methods for deleting documents as discussed in Table 1-10.

Table 1-10. MongoCollection<TDocument> Methods to Delete Documents

Method	Description
deleteMany(Bson filter)	Deletes all documents from a collection using a query filter and returns result as DeleteResult. The DeleteResult contains information about number of documents deleted, and whether the delete was acknowledged.
deleteOne(Bson filter)	Deletes one document based on a query filter and returns result as DeleteResult.
findOneAndDelete(Bson filter)	Finds a document based on a query filter and deletes the document. Returns the deleted document.
findOneAndDelete(Bson filter, FindOneAndDeleteOptions options)	Finds a document based on a query filter and the FindOneAndDeleteOptions options and deletes the document. Returns the deleted document. FindOneAndDeleteOptions options include the maximum time to find and delete, the sort criteria, and which document fields to return.

1. In the DeleteDBDocument application create a MongoClient client as discussed previously. Create a MongoDB instance for the local database from the MongoClient instance and create a MongoCollection<TDocument> instance for the catalog collection from the MongoDB instance.

```
MongoClient mongoClient = new MongoClient(Arrays.asList(new
ServerAddress("localhost", 27017)));
MongoDatabase db = mongoClient.getDatabase("local");
MongoCollection<Document> coll = db.getCollection("catalog");
```

2. Create and add four Document instances using the model class Catalog to set the Document fields.

```
Document catalog = new Document("catalogId", "catalog1")
    .append("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013")
    .append("title", "Engineering as a Service")
    .append("author", "David A. Kelly");
coll.insertOne(catalog);

catalog = new Document("catalogId", "catalog2")
    .append("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013")
    .append("title", "Quintessential and Collaborative")
    .append("author", "Tom Haunert");
coll.insertOne(catalog);
```

```

catalog = new Document("catalogId", "catalog3")
    .append("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013");
coll.insertOne(catalog);

catalog = new Document("catalogId", "catalog4")
    .append("journal", "Oracle Magazine")
    .append("publisher", "Oracle Publishing")
    .append("edition", "November December 2013");
coll.insertOne(catalog);

```

- As an example of using the `deleteOne(Bson filter)` method delete the document with catalogId as catalog1.

```
DeleteResult result = coll.deleteOne(new Document("catalogId", "catalog1"));
```

- The `deleteOne()` method returns a `DeleteResult` object. Output the number of documents deleted using the `getDeletedCount()` method of `DeleteResult`.

```
System.out.println("Number of documents deleted: "
+ result.getDeletedCount());
```

- As an example of using the `findOneAndDelete(Bson filter)` method delete the document with catalogId as catalog2. The `findOneAndDelete()` method returns the document deleted. Output the deleted document.

```
Document documentDeleted = coll.findOneAndDelete(new
Document("catalogId", "catalog2"));
System.out.println("Document deleted: " + documentDeleted);
```

- As an example of using the `deleteMany(Bson filter)` method delete all the remaining documents by providing a `Document` object without a specified key/value pair as a method argument. Subsequently, output the number of documents deleted using the `getDeletedCount()` method in `DeleteResult`.

```
DeleteResult result = coll.deleteMany(new Document());
System.out.println("Number of documents deleted: " + result.
getDeletedCount());
```

- To verify that the document/s got deleted, find all the documents using the `find()` method and output the key/value pairs in each of the documents as discussed before. The `DeleteDBDocument` application is listed below.

```

package mongodb;

import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;

```

```

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.DeleteResult;

public class DeleteDBDocument {

    public static void main(String[] args) {

        MongoClient mongoClient = new MongoClient(
            Arrays.asList(new ServerAddress("localhost", 27017)));

        MongoDatabase db = mongoClient.getDatabase("local");

        MongoCollection<Document> coll = db.getCollection("catalog");
        Document catalog = new Document("catalogId", "catalog1")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Engineering as a Service")
            .append("author", "David A. Kelly");
        coll.insertOne(catalog);

        catalog = new Document("catalogId", "catalog2")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013")
            .append("title", "Quintessential and Collaborative")
            .append("author", "Tom Haunert");
        coll.insertOne(catalog);

        catalog = new Document("catalogId", "catalog3")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013");
        coll.insertOne(catalog);

        catalog = new Document("catalogId", "catalog4")
            .append("journal", "Oracle Magazine")
            .append("publisher", "Oracle Publishing")
            .append("edition", "November December 2013");
        coll.insertOne(catalog);

        DeleteResult result = coll.deleteOne(
            new Document("catalogId", "catalog1"));

        System.out.println("Number of documents deleted: "
            + result.getDeletedCount());
    }
}

```

```
Document documentDeleted = coll
    .findOneAndDelete(new Document("catalogId", "catalog2"));
System.out.println("Document deleted: " + documentDeleted);

result = coll.deleteMany(new Document());

System.out.println("Number of documents deleted: "
    + result.getDeletedCount());

FindIterable<Document> iterable = coll.find();

String documentKey = null;

for (Document document : iterable) {
    Set<String> keySet = document.keySet();
    Iterator<String> iter = keySet.iterator();
    while (iter.hasNext()) {
        documentKey = iter.next();
        System.out.println(documentKey);
        System.out.println(document.get(documentKey));
    }
}
mongoClient.close();
}

}
```

8. Before running the application drop the catalog collection with db.catalog.drop() command in mongo shell. To run the DeleteDBDocument application, right-click on the DeleteDBDocument.java file in Package Explorer and select Run As ➤ Java Application as shown in Figure 1-18.

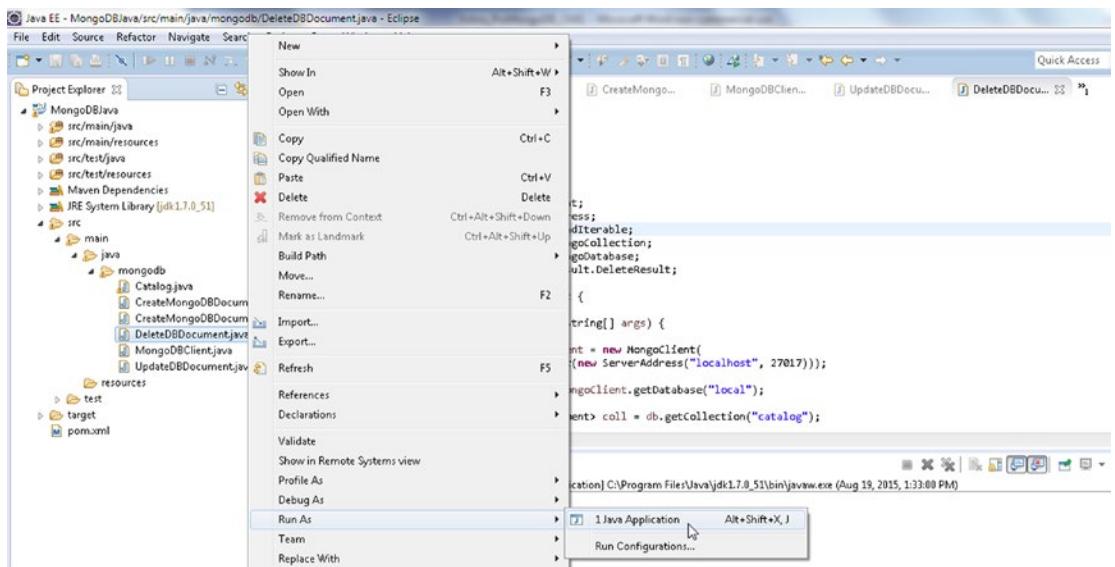


Figure 1-18. Running the DeleteDBDocument.java Application

The output from DeleteDBDocument.java application is shown in Figure 1-19.

```
<terminated> DeleteDBDocument [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 19, 2015, 1:33:00 PM)
INFO: Opened connection [connectionId{localValue:2, serverValue:15}] to localhost:27017
Number of documents deleted: 1
Document deleted: Document{{_id=55d4e7fed292641860c77855, catalogId=catalog2, journal=Oracle Magazine, publisher=Oracle Publishing, edition=November December 2013, title=Quintessential and Collaborative, author=Tom Haunert}}
Number of documents deleted: 2
Aug 19, 2015 1:33:02 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Closed connection [connectionId{localValue:2, serverValue:15}] to localhost:27017 because the pool has been closed.
```

Figure 1-19. Output from DeleteDBDocument.java Application

As the `deleteOne()` method deletes one document, the number of documents indicated to have been deleted subsequent to the invocation of the `deleteOne()` method is output as 1. The document deleted with the `findOneAndDelete()` method is output. The delete count for the `deleteMany()` method is output as 2, which is the number of documents in the catalog collection after having deleted 2 of the 4 documents added in the `DeleteDBDocument` application.

Summary

In this chapter we used the MongoDB Java driver to access MongoDB server and add documents to the database. Subsequently we fetched the documents from the database and also updated and deleted the documents. In this chapter we also introduced the Mongo shell. In the next chapter we shall discuss the Mongo shell.

CHAPTER 2



Using the Mongo Shell

The MongoDB data storage structures are similar to those of a SQL relational database. A MongoDB database is similar to an SQL database. A table in an SQL database is a collection in MongoDB. A row in an SQL database is a document in MongoDB. A column in an SQL database is a field in MongoDB. MongoDB distribution includes an interactive shell called the Mongo shell. The Mongo shell provides database commands of different kinds including aggregation commands (for example, the count command finds the number of documents in a collection), collection commands (for example, the create command creates a collection), and administration commands (for example, the copydb command copies a database instance). The Mongo shell provides various JavaScript Mongo shell helper methods for data operations and administration. Some of the database commands have the equivalent Mongo shell helper methods, while others don't. In this chapter we shall discuss how to access the Mongo shell and run database commands and JavaScript helper methods on a database, collection, or document.

This chapter includes the following topics:

- Getting started
- Using databases
- Using collections
- Using documents

Getting Started

In the following subsections we shall set up the environment including installing the required software. We shall start Mongo shell and connect with MongoDB server. And we will also discuss running a command in Mongo shell.

Setting Up the Environment

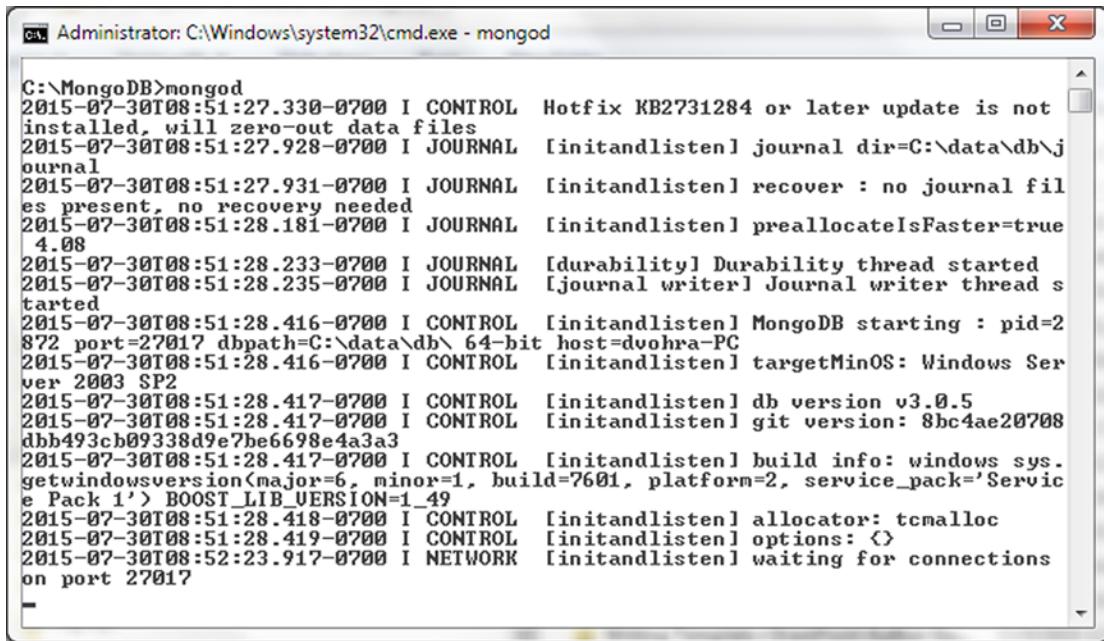
Download and install the following software if not already installed from Chapter 1:

- MongoDB (3.0.5) for Windows 64-bit is used in this chapter. Download binary distribution from <http://www.mongodb.org/downloads>.

Double-click on the MongoDB binary distribution to install MongoDB. Add the bin directory of the MongoDB installation to the PATH environment. Create a directory C:\data\db for the MongoDB data. Start the MongoDB server with the following command in a Command window.

```
>mongod
```

The MongoDB server gets started as shown in Figure 2-1.



```
C:\MongoDB>mongod
2015-07-30T08:51:27.330-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-07-30T08:51:27.928-0700 I JOURNAL [initandlisten] journal dir=C:\data\db\j
ournal
2015-07-30T08:51:27.931-0700 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-07-30T08:51:28.181-0700 I JOURNAL [initandlisten] preallocateIsFaster=true
4.08
2015-07-30T08:51:28.233-0700 I JOURNAL [durability] Durability thread started
2015-07-30T08:51:28.235-0700 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-07-30T08:51:28.416-0700 I CONTROL [initandlisten] MongoDB starting : pid=2
872 port=27017 dbpath=C:\data\db\ 64-bit host=dvohra-PC
2015-07-30T08:51:28.416-0700 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-07-30T08:51:28.417-0700 I CONTROL [initandlisten] db version v3.0.5
2015-07-30T08:51:28.417-0700 I CONTROL [initandlisten] git version: 8bc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-07-30T08:51:28.417-0700 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-07-30T08:51:28.418-0700 I CONTROL [initandlisten] allocator: tcmalloc
2015-07-30T08:51:28.419-0700 I CONTROL [initandlisten] options: <>
2015-07-30T08:52:23.917-0700 I NETWORK [initandlisten] waiting for connections
on port 27017
```

Figure 2-1. Starting MongoDB

Starting the Mongo Shell

Open a new Command window. Start the Mongo shell with the following command.

```
>mongo
```

The Mongo shell gets started and gets connected to the test database by default as shown in Figure 2-2. By default the Mongo shell connects to MongoDB on localhost on port 27017.



```
C:\MongoDB>mongo
2015-07-30T08:53:09.672-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
> -
```

Figure 2-2. Starting Mongo Shell

A message similar to "Hotfix KB2731284 or later update is not installed, will zero-out data files" might get generated. One of the reasons for the message is that MongoDB is installed in a directory such that the directory path has spaces, for example, directory path C:\Program Files\MongoDB\Server\3.0. The message may be ignored if directory path with spaces is the cause of the message. If the reason for the message is that the C:\data\db directory has not been created, create the directory C:\data\db.

To start the Mongo shell without connecting to any database, run the following command. Before running the following command, open a new command window as we already connected once using the mongo command.

```
>mongo --nodb
```

Subsequently a connection may be opened to MongoDB server using the JavaScript `Mongo()` constructor, which by default connects to localhost at port 27017.

```
>connection=new Mongo()
```

The `getDB()` method in Mongo shell may be used to set the database as shown in Figure 2-3.

```
db=connection.getDB("test")
```

```
C:\MongoDB>mongo --nodb
2015-07-30T08:56:30.928-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: --nodb
> connection=new Mongo()
connection to 127.0.0.1
> db=connection.getDB("test")
test
>
```

Figure 2-3. Starting Mongo Shell without Connecting to a Database at First

As a result of the preceding commands the Mongo shell is connected to MongoDB server on the default host localhost and on the default port 27017 with the test database in use, which is the same if just the `mongo` command is run. Connecting using the `Mongo()` constructor is useful if connecting to a non-default host and port.

Alternatively, the `connect` method may be used to connect to MongoDB in Mongo shell. After starting Mongo shell with `--nodb` option, connect to MongoDB on localhost and port 27017 and set database as `test` with the following command.

```
db = connect("localhost:27017/test");
```

The `connect()` method gets connected to MongoDB test database as shown in Figure 2-4.

The screenshot shows a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongo -nodb". The command "mongo -nodb" is run, and the MongoDB shell version 3.0.5 connects to the database "test" on the local host at port 27017. A message from Microsoft Hotfix KB2731284 is displayed.

```
C:\MongoDB>mongo -nodb
2015-07-30T08:58:15.579-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: -nodb
> db = connect("localhost:27017/test")
connecting to: localhost:27017/test
test
>
```

Figure 2-4. Using the `connect()` Method to Connect to MongoDB

Running a Command or Method in Mongo Shell

The following kinds of commands and methods may be run in Mongo shell.

- The database commands.
- Mongo shell JavaScript helper methods
- Mongo shell help methods

The difference between the JavaScript helper methods and the help methods is that the JavaScript helper methods make use of JavaScript. The database commands have the BSON document format consisting of key/value pairs with the first key being the name of the command and subsequent key/value pairs being the command options. For example, the `create` command, which is used to create a collection, has the following syntax.

```
{ create: <collection_name>, option1:value, option2:value, option3:value,..optionN:value }
```

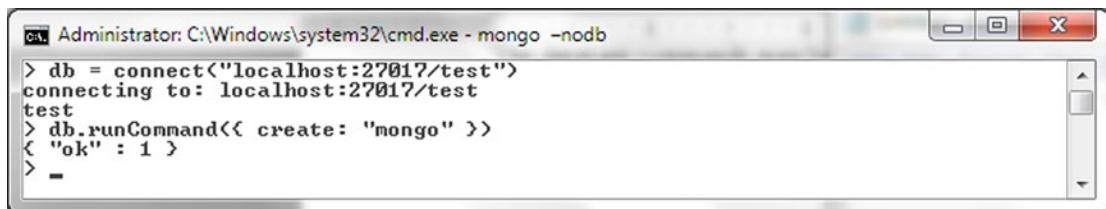
The `create` command provides the following options (only the main options are discussed).

- `capped`. Set to boolean value (true or false) to specify if the collection is a capped collection. Default value is false. If true is set, the `size` option is also required. A capped collection is a fixed size collection in which the earlier documents are overwritten when the maximum size is reached.
- `autoIndexId`. Set to boolean value (true or false) to specify if an automatic index is to be created on the `_id` field. The default value is true.
- `size`. The maximum size in bytes of a capped collection. Required for a capped collection.
- `max`. Maximum number of documents in a capped collection. The size setting takes precedence over the max setting. For example, if `size` is 3 and `max` is 4, the maximum number of documents is 3.

The database commands may be run using the Mongo shell helper method `db.runCommand()`. For example the `create` command may be run to create a collection "mongo" using the following helper/wrapper JavaScript method in Mongo shell.

```
db.runCommand({ create: "mongo" })
```

A command response is a JSON document with at least the `ok` field, which indicates if the command succeeded as shown in Figure 2-5. A value of 1 indicates the command succeeded and a value of 0 indicates the command failed.



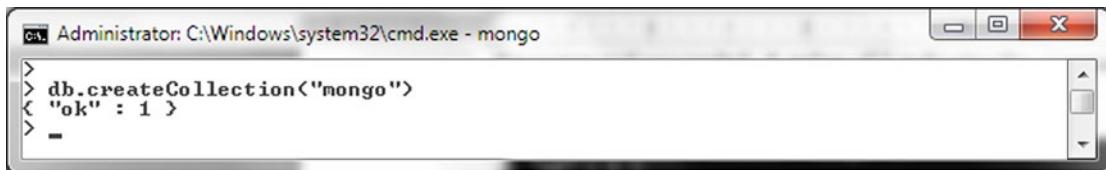
```
Administrator: C:\Windows\system32\cmd.exe - mongo -nodb
> db = connect("localhost:27017/test")
connecting to: localhost:27017/test
test
> db.runCommand({ create: "mongo" })
{ "ok" : 1 }
> -
```

Figure 2-5. Using the `db.runCommand()` Helper Method

The `create` command is one of those commands that have an equivalent JavaScript shell helper method, the `db.createCollection()` method. The `mongo` collection could have been created as follows.

```
db.createCollection("mongo")
```

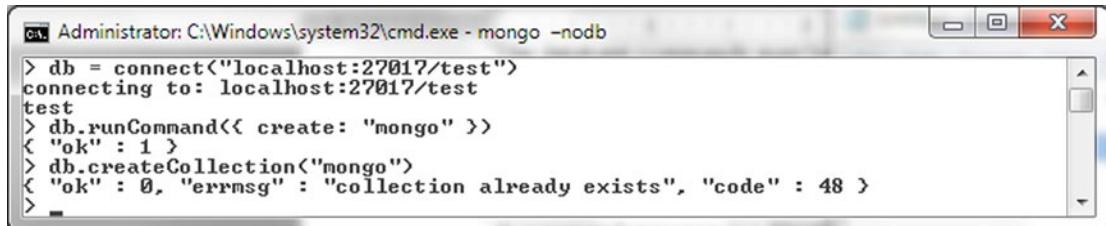
The preceding command returns a JSON document with `ok` field set to 1, which indicates the command succeeded as shown in Figure 2-6.



```
Administrator: C:\Windows\system32\cmd.exe - mongo
>
> db.createCollection("mongo")
{ "ok" : 1 }
> -
```

Figure 2-6. Using the `db.createCollection()` Helper Method

If the preceding command is to be run subsequent to the `db.runCommand` to create the `mongo` collection, the collection must be deleted using the `db.mongo.drop()` command to avoid the “collection already exists” error message, which is shown in Figure 2-7. (See “Dropping a Collection” later in this chapter.)



```
Administrator: C:\Windows\system32\cmd.exe - mongo -nodb
> db = connect("localhost:27017/test")
connecting to: localhost:27017/test
test
> db.runCommand({ create: "mongo" })
{ "ok" : 1 }
> db.createCollection("mongo")
{ "ok" : 0, "errmsg" : "collection already exists", "code" : 48 }
> -
```

Figure 2-7. The “collection already exists” Error Message

A complete reference of the database commands is available at <http://docs.mongodb.org/v3.0/reference/command/>, and a complete reference of Mongo shell helper methods is available at <http://docs.mongodb.org/v3.0/reference/method/>.

JavaScript methods may also be run using a JavaScript file. For example, copy the following JavaScript to a file `connection.js` in the directory `C:\MongoDB` from which the command is to be run.

```
connection = new Mongo();
db = connection.getDB("test");
printjson(db.getCollectionNames());
```

From the command line run the JavaScript file with the following command.

```
>mongo connection.js
```

The JavaScript gets evaluated and the output gets generated, which is a listing of collection names for the example JavaScript as shown in Figure 2-8.

```
C:\MongoDB>mongo connection.js
2015-07-30T09:08:38.459-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
[ "mongo", "products", "system.indexes", "users" ]
C:\MongoDB>
```

Figure 2-8. Running a JavaScript Script

The Mongo shell also provides some help methods to get information about the databases, collections, and documents. Some of the help methods are listed in following table, Table 2-1.

Table 2-1. Help Methods

Help Command	Description
show dbs	Lists the databases on the server.
use <db>	Selects a database. A database must be selected before any operation on a collection or a document may be performed.
show collections	Lists all the collections in a database.

Using Databases

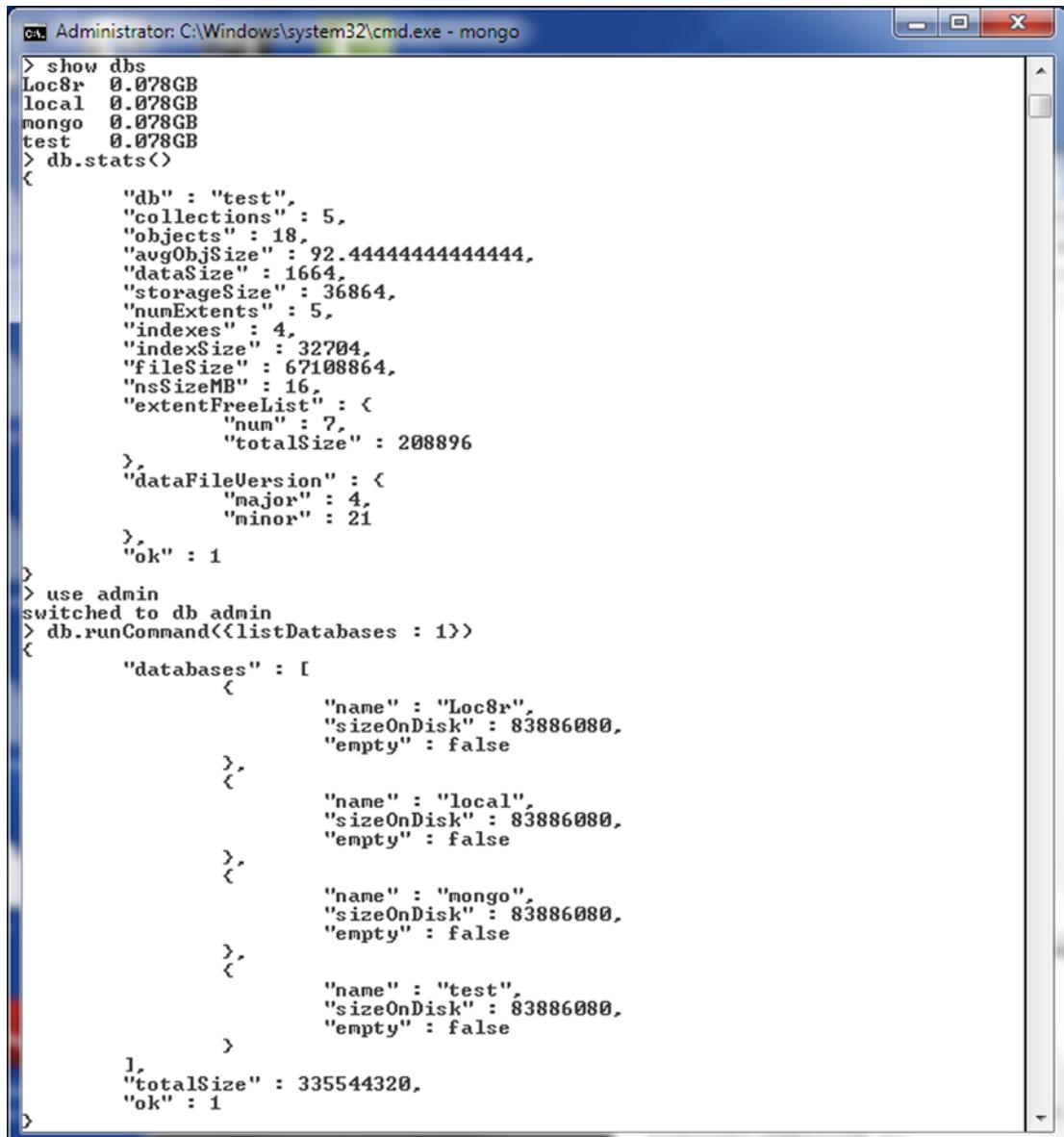
In the following subsections we shall discuss getting information about a database, creating a database, and dropping a database.

Getting Databases Information

The Mongo shell has a variable called db, which references the current database. By default when the Mongo shell is started using the mongo command the test database becomes the current database as discussed in an earlier section. The show dbs command lists all the databases on the server. The db.stats() Mongo shell helper method lists the stats on the current database. The listDatabases command lists all the databases including the total size and size for each database and if the database is empty. Some commands – the administrative commands – such as the listDatabases command must be run on the admin database.

```
>show dbs
>db.stats()
>use admin
>db.runCommand({listDatabases : 1})
```

The output from the preceding commands gets displayed in the Mongo shell as shown in Figure 2-9.



A screenshot of a Windows command prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongo". The window displays the results of MongoDB shell commands. The first part shows database statistics for the "test" database, including collection count, object count, average object size, data size, storage size, index counts, file size, and extent free list details. The second part shows the result of the `listDatabases` command, which lists all databases in the system, their names, sizes on disk, and whether they are empty. The "test" database is listed at the end with a total size of 335544320 bytes.

```
> show dbs
Loc8r 0.078GB
local 0.078GB
mongo 0.078GB
test 0.078GB
> db.stats()
{
  "db" : "test",
  "collections" : 5,
  "objects" : 18,
  "avgObjSize" : 92.44444444444444,
  "dataSize" : 1664,
  "storageSize" : 36864,
  "numExtents" : 5,
  "indexes" : 4,
  "indexSize" : 32704,
  "fileSize" : 67108864,
  "nsSizeMB" : 16,
  "extentFreeList" : {
    "num" : 7,
    "totalSize" : 208896
  },
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 21
  },
  "ok" : 1
}
> use admin
switched to db admin
> db.runCommand({listDatabases : 1})
{
  "databases" : [
    {
      "name" : "Loc8r",
      "sizeOnDisk" : 83886080,
      "empty" : false
    },
    {
      "name" : "local",
      "sizeOnDisk" : 83886080,
      "empty" : false
    },
    {
      "name" : "mongo",
      "sizeOnDisk" : 83886080,
      "empty" : false
    },
    {
      "name" : "test",
      "sizeOnDisk" : 83886080,
      "empty" : false
    }
  ],
  "totalSize" : 335544320,
  "ok" : 1
}
```

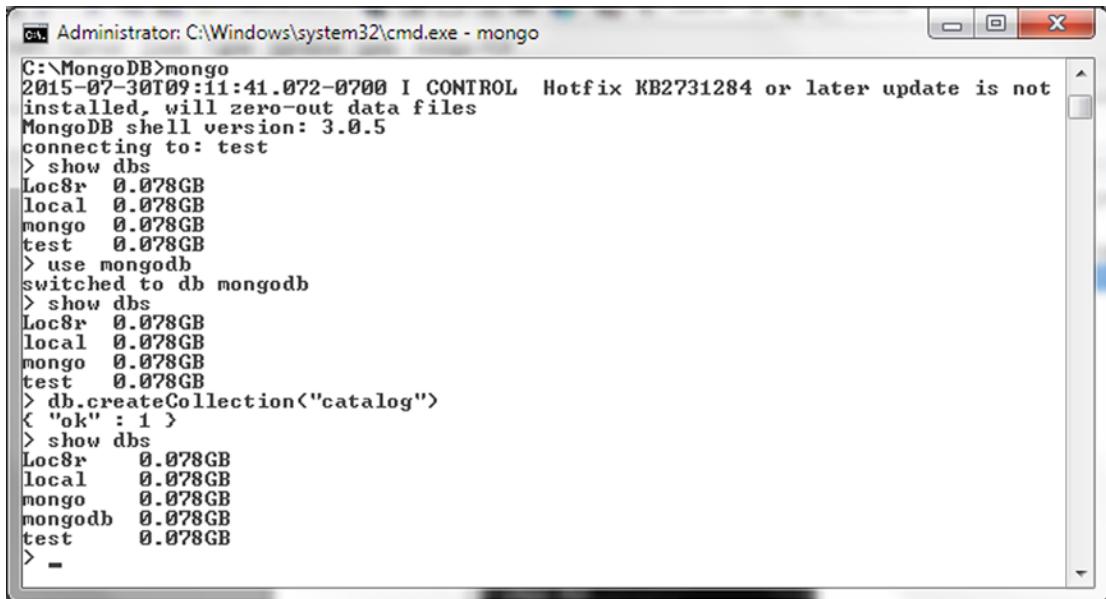
Figure 2-9. Getting Databases Information

Creating a Database Instance

A MongoDB database instance is created implicitly when a command is sent to the database instance and an operation is performed such as creating a collection. The `use <db>` command may be used to select the current database even if the database does not already exist, but the `use <db>` command does not create a nonexistent database. To create the database an operation must be run on the database. For example, list all databases with the `show dbs` command. Subsequently select the current database as a nonexistent database `mongodb` using the `use mongodb` command. And subsequently create a collection called `catalog` using the `db.createCollection (name,options)` helper method.

```
>show dbs
>use mongodb
>show dbs
>db.createCollection("catalog")
>show dbs
```

If the `show dbs` command is run after the `use mongodb` command the `mongodb` database does not get listed, but if the `show dbs` command is run after the `db.createCollection()` command the `mongodb` database gets listed as shown in Figure 2-10.



The screenshot shows a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongo". The command `mongo` is run, followed by several `show dbs` commands. The first two `show dbs` commands show the databases `Loc8r`, `local`, `mongo`, and `test`, each with a size of `0.078GB`. The third `show dbs` command, after running `> use mongodb`, shows the same four databases plus the newly created `catalog` database, all with a size of `0.078GB`.

```
C:\MongoDB>mongo
2015-07-30T09:11:41.072-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
> show dbs
Loc8r 0.078GB
local 0.078GB
mongo 0.078GB
test 0.078GB
> use mongodb
switched to db mongodb
> show dbs
Loc8r 0.078GB
local 0.078GB
mongo 0.078GB
test 0.078GB
> db.createCollection("catalog")
{
  "ok" : 1
}
> show dbs
Loc8r 0.078GB
local 0.078GB
mongo 0.078GB
mongodb 0.078GB
test 0.078GB
> -
```

Figure 2-10. Creating a Database Instance

The `db.copyDatabase(fromdb, todb, fromhost, username, password, mechanism)` command may be used to copy a database to another database. The target database gets created. For example, copy the database `local` to a new database called `catalog`.

```
db.copyDatabase('local', 'catalog')
```

If the `show dbs` command is run before and after the `db.copyDatabase()` command, the command run after it lists the `catalog` database as shown in Figure 2-11.

```
> show dbs
Loc8r 0.078GB
local 0.078GB
mongo 0.078GB
mongodb 0.078GB
test 0.078GB
> db.copyDatabase('local', 'catalog')
{ "ok" : 1 }
> show dbs
Loc8r 0.078GB
catalog 0.078GB
local 0.078GB
mongo 0.078GB
mongodb 0.078GB
test 0.078GB
> -
```

Figure 2-11. Copying a Database Instance

Dropping a Database

To remove the current database use the `db.dropDatabase()` method. The current database is the database set with the `use <db>` command. Even after dropping a database the current database name is not changed, and if a new collection is created, it is created in a database of the same name as the dropped database using new data files. For example list all databases using the `show dbs` command. Subsequently set the current database as one of the databases listed, for example, database `catalog`. The database chosen for deletion could be different and does not have to be called `catalog`. Subsequently invoke the `db.dropDatabase()` method to drop the current database, which is the database `mongo`. If subsequently the `show dbs` command is invoked the `catalog` database does not get listed.

```
>show dbs
>use catalog
>db.dropDatabase()
>show dbs
```

The `db.dropDatabase()` method returns a document with fields “dropped” and “ok.” The “dropped” field value is the database dropped, and the “ok” field value of 1 indicates the method returned without error as shown in Figure 2-12.

```
> show dbs
Loc8r 0.078GB
catalog 0.078GB
local 0.078GB
test 0.078GB
> use catalog
switched to db catalog
> db.dropDatabase()
{ "dropped" : "catalog", "ok" : 1 }
> show dbs
Loc8r 0.078GB
local 0.078GB
test 0.078GB
> -
```

Figure 2-12. Dropping a Database

To quit the Mongo shell run the `quit()` command or the `exit` command.

```
>quit()
```

The Mongo shell session gets ended as shown in Figure 2-13.

Figure 2-13. Ending a Mongo Shell Session

Using Collections

In the following subsections we shall create a collection and subsequently drop a collection from the Mongo shell.

Creating a Collection

The `create` command is used to create a collection. We already discussed the `create` command to create a collection (see “Running a Command or Method in mongo Shell”). The `show collections` command may be used to list the collections in a database.

```
>use test
>show collections
```

The collections in the database get listed as shown in Figure 2-14.

Figure 2-14. Listing Collections with `show collections`

The collections listed could be different for different users but the `system.indexes` collection should get listed regardless of the user. A collection gets listed only after it has been created. For example, list the collections before and after running the `db.createCollection()` method to create a collection `catalog` in the `test` database. If the `catalog` collection has already been created, drop the collection with `db.catalog.drop()`

```
>use test
>db.catalog.drop()
>show collections
>db.createCollection("catalog")
>show collections
```

The `show collections` command runs after the `db.createCollection()` method lists the catalog collection as shown in Figure 2-15. The other collections listed could be different for different users.

```
> use test
switched to db test
> show collections
mongo
products
system.indexes
users
> db.createCollection("catalog")
{ "ok" : 1 }
> show collections
catalog
mongo
products
system.indexes
users
> _
```

Figure 2-15. Running `show collections` after Creating a Collection

A *capped* collection is a collection with a fixed size, similar to a circular list. When a capped collection is full the data gets overwritten. Data cannot be deleted from a capped collection. For example, create a capped collection catalog with auto indexing and 64 KB size with maximum number of documents as 1000. Before creating the capped catalog collection, drop the collection if created previously using the `db.catalog.drop()` command. (Dropping is discussed in more detail in the following section.)

```
>use test
>db.catalog.drop()
>db.createCollection("catalog", {capped: true, autoIndexId: true, size: 64 * 1024, max: 1000} )
```

A response with `ok` field as 1 indicates the capped collection gets created as shown in Figure 2-16.

```
> db.catalog.drop()
true
> db.createCollection("catalog", {capped: true, autoIndexId: true, size: 64 * 1024, max: 1000} )
{ "ok" : 1 }
> _
```

Figure 2-16. Creating a Capped Collection

The create command may also be run using the `db.runCommand()` method. As before, if the catalog collection already exists drop it first. The `use test` command is not required to be rerun if already using the test database.

```
>use test
>db.catalog.drop()
>db.runCommand( { create: "catalog", capped: true, size: 64 * 1024, max: 1000 } )
```

The capped collection gets created as indicated by the "ok": 1 response in Figure 2-17.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.drop()
true
> db.runCommand( { create: "catalog", capped: true, size: 64 * 1024, max: 1000 } )
{
  "ok" : 1
}
>
```

Figure 2-17. Creating a Capped Collection with `runCommand()`

A collection must not already exist or an error gets generated. For example, create the catalog collection when it already exists. The “collection already exists” error message gets output as shown in Figure 2-18.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.runCommand( { create: "catalog", capped: true, size: 64 * 1024, max: 1000 } )
{
  "ok" : 0,
  "errmsg" : "collection already exists",
  "code" : 48
}
>
```

Figure 2-18. The “collection already exists” Error Message

The number of documents in a collection may be listed with the following helper method `db.collection.count()`. For example, the following method lists the document count in the catalog collection.

```
>db.catalog.count()
```

An output of 0 indicates no documents in the catalog collection have been added, as shown in Figure 2-19.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.count()
0
> -
```

Figure 2-19. Listing the Document Count

The stats on a collection may be listed with the following `db.collection.stats()` method. For example, the following method lists the stats on the `catalog` collection.

```
>db.catalog.stats()
```

The collection stats output include the collection namespace in the format `<database>.collection`, the document count, if the collection is capped, the storage size, and the maximum number of documents and more as shown in Figure 2-20.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.stats()
{
    "ns" : "test.catalog",
    "count" : 0,
    "size" : 0,
    "numExtents" : 1,
    "storageSize" : 65536,
    "lastExtentSize" : 65536,
    "paddingFactor" : 1,
    "paddingFactorNote" : "paddingFactor is unused and unmaintained in 3.0. It remains hard coded to 1.0 for compatibility only.",
    "userFlags" : 1,
    "capped" : true,
    "max" : 1000,
    "maxSize" : 65536,
    "nindexes" : 1,
    "totalIndexSize" : 8176,
    "indexSizes" : {
        "_id_" : 8176
    },
    "ok" : 1
}
> -
```

Figure 2-20. Listing Stats with the `stats()` Method

A collection may be renamed with the `db.collection.renameCollection(target, dropTarget)` method. By default the `dropTarget` boolean is `false` indicating if a collection by the same name as the new name of the collection already exists should the target collection be dropped. For example, rename the `mongo` collection to `mongodb`.

```
>db.mongo.renameCollection('mongodb', false)
```

A response of `"ok": 1` indicates the `mongo` collection gets renamed as shown in Figure 2-21. If you run the `show collections` command after the renaming, it lists the renamed collection `mongodb` instead of the `mongo` collection listed prior to renaming.

```
>
> show collections
catalog
mongo
products
system.indexes
users
> db.mongo.renameCollection('mongodb', false)
{ "ok" : 1 }
> show collections
catalog
mongodb
products
system.indexes
users
> -
```

Figure 2-21. Renaming a Collection

Dropping a Collection

To remove a collection from the database invoke the `db.collection.drop()` method. For example, invoke the `show collections` command to list all collections. Subsequently drop the `catalog` collection.

```
>show collections
>db.catalog.drop()
>show collections
```

If the `show collections` command is invoked subsequent to dropping the `catalog` collection, the `catalog` collection is not listed as shown in Figure 2-22.

```
>
> show collections
catalog
catalog-collection
catalogColl
catalogReplace
catalog_Replace
catalog_coll
startup_log
system.indexes
wlslog
> db.catalogColl.drop()
true
> db.catalog_Replace.drop()
true
> db.catalog_coll.drop()
true
> db.catalogReplace.drop()
true
> show collections
catalog
catalog-collection
startup_log
system.indexes
wlslog
> -
```

Figure 2-22. Dropping a Collection

In most examples in this chapter we have dropped a previously created collection and created the same collection for the example to demonstrate the effect of the command used in the example. Running commands with a previously created collection may not fully demonstrate the effect of a command.

Using Documents

In the following subsections we shall discuss adding a document, adding documents in a batch, querying a document, updating a document, and removing a document.

Adding a Document

In this section we shall use the `db.collection.insert()` JavaScript method to add a document to MongoDB from Mongo shell. The `insert()` helper method has a different syntax for different versions of MongoDB with some new features added to latter versions as listed in Table 2-2.

Table 2-2. Insert Helper Method Syntax

MongoDB Version	insert Method Syntax	Description
2.6 and later	<code>db.collection.insert(<document or array of documents>,{writeConcern:<document>,ordered:<boolean>})</code>	Added the <code>writeConcern</code> and <code>ordered</code> options – both of which are optional. The <code>writeConcern</code> option provides different levels of guarantees on the success of an insert for “safe writes.” The <code>ordered</code> option takes a Boolean value and performs an ordered insert of documents with the default being false. The <code>insert</code> method returns a <code>WriteResult</code> object for a single document insert and a <code>BulkWriteResult</code> object for an array of documents.
2.2	<code>db.collection.insert(<document or array of documents>)</code>	Supports adding a single document or an array of documents.
2.04	<code>db.collection.insert(<document>)</code>	Supports adding only one document. The single document may contain subdocuments.

To add a document, complete the following steps:

1. Drop the `catalog` collection if it already exists before adding a new document or array of documents.

```
>db.catalog.drop()
```

2. Create the BSON document construct to add.

```
>use catalog
>doc1 = {"catalogId" : "catalog1", "journal" : 'Oracle Magazine',
  "publisher" : 'Oracle Publishing', "edition" : 'November December
  2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
```

3. Invoke the db.collection.insert() method on the document using the catalog collection.

```
>db.catalog.insert(doc1)
```

4. Subsequently run the db.collection.find() method to find the documents in the catalog collection.

```
>db.catalog.find()
```

The single document added gets listed and includes the _id field. The _id field, which was not specified in the document JSON construct, gets added automatically before the document is added to the database.

A WriteResult object returned by the insert() method includes the field nInserted to indicate the number of documents added. With one document added nInserted field value is 1 as shown in Figure 2-23.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> use catalog
switched to db catalog
> doc1 = {"catalogId" : "catalog1", "journal" : 'Oracle Magazine', "publisher" :
  "Oracle Publishing", "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
{
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
}
> db.catalog.insert(doc1)
WriteResult({ "nInserted" : 1 })
> db.catalog.find()
[ { "_id" : ObjectId("55ba5291403cc01aa5b078db"), "catalogId" : "catalog1", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" } ]
> -
```

Figure 2-23. Using the Insert Method to Add a Document

Next, we shall add a BSON document in which the `_id` field is specified in the BSON document construct.

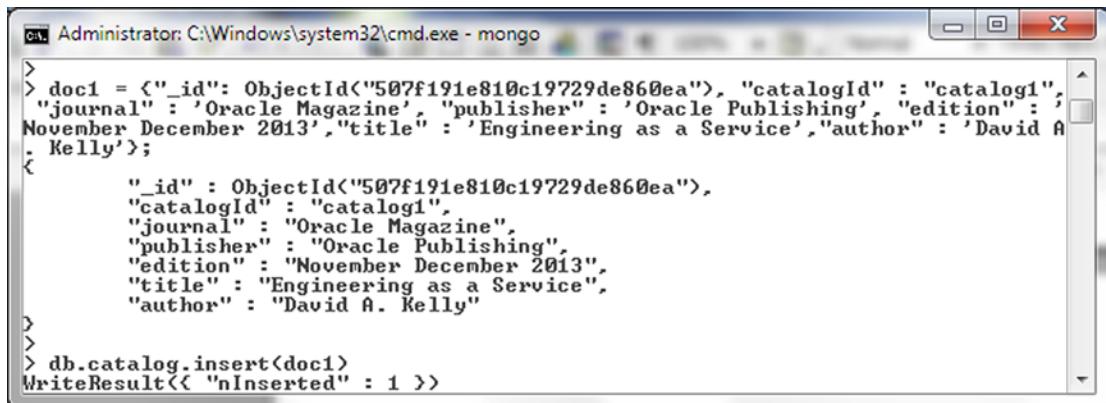
1. Specify the same document as earlier but with an additional field `_id`. The value of the `_id` field must be an `ObjectId` object.

```
> doc1 = { "_id": ObjectId("507f191e810c19729de860ea"), "catalogId"
  : "catalog1", "journal" : 'Oracle Magazine', "publisher" : 'Oracle
  Publishing', "edition" : 'November December 2013', "title" :
  'Engineering as a Service', "author" : 'David A. Kelly'};
```

2. Add the document using the `db.collection.insert` method.

```
> db.catalog.insert(doc1)
```

As before, the `insert()` method returns a `WriteResult` object with the `nInserted` field value as 1 indicating that one document has been added as shown in Figure 2-24.



```
>
> doc1 = { "_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog1",
  "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" :
  "November December 2013", "title" : "Engineering as a Service", "author" :
  "David A. Kelly"};
<
  "_id" : ObjectId("507f191e810c19729de860ea"),
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
>
> db.catalog.insert(doc1)
WriteResult{<< "nInserted" : 1 >>}
```

Figure 2-24. Adding a BSON Document Including the `_id` field

3. Subsequently run the `db.catalog.find()` method to find the documents in the database. The document added gets listed and has the `_id` field as specified in the document added as shown in Figure 2-25.



```
>
> db.catalog.find()
{ "_id" : ObjectId("55ba5291403cc01aa5b078db"), "catalogId" : "catalog1", "jour
nal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "Novembe
r December 2013", "title" : "Engineering as a Service", "author" : "David A. Kel
ly" }
{ "_id" : ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog1", "jour
nal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "Novembe
r December 2013", "title" : "Engineering as a Service", "author" : "David A. Kel
ly" }
> -
```

Figure 2-25. Listing Documents with `_id` Field

For any version of MongoDB the `_id` field value should be unique among the documents in a collection. To demonstrate, add the following document, which has the `_id` field set to `ObjectId("507f191e810c19729de860ea")`.

```
>doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
>db.catalog.insert(doc2)
```

Add another document with the `_id` field set to the same `ObjectId` value.

```
>doc3 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 3};
>db.catalog.insert(doc3)
```

A duplicate key error gets generated as shown in Figure 2-26.

```
> doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
{
    "_id" : ObjectId("507f191e810c19729de860ea"),
    "catalogId" : 2,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013"
}
>
> db.catalog.insert(doc2)
WriteResult({
    "nInserted" : 1,
    "writeError" : {
        "code" : 11000,
        "errmsg" : "E11000 duplicate key error index: catalog.catalog.$_id_ dup key: { : ObjectId('507f191e810c19729de860ea') }"
    }
})
>
```

Figure 2-26. Duplicate Key Error

Adding a Batch of Documents

The support for adding an array of documents was added in MongoDB 2.2. In the pre-2.2 version only a single document could be added with a single invocation of the `db.collection.insert()` method. To add a batch of documents in MongoDB 3.0.5, specify or create JSON for two documents to be added including the `_id` fields.

```
>doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 'catalog1',
"journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
>doc2 = {"_id": ObjectId("53fb4b08d17e68cd481295d5"), "catalogId" : 'catalog2',
"journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
```

Drop the catalog collection and invoke the db.catalog.insert() method using an array of documents with collection as catalog.

```
>db.catalog.drop()
>db.catalog.insert([doc1, doc2])
```

The insert() method returns a BulkWriteResult object, which includes a field nInserted, which signifies the number of documents added. The two documents in the array are added as two distinct documents as indicated by the nInserted field value as 2. Subsequently run the db.collection.find() method to find the documents in the catalog collection.

```
>db.catalog.find()
```

The collection method find() lists two distinct documents as shown in Figure 2-27.

```
> db.catalog.drop()
> db.catalog.insert([{"_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}])
>
> db.catalog.insert([{"_id": ObjectId("53fb4b08d17e68cd481295d5"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}, {"_id": ObjectId("53fb4b08d17e68cd481295d5"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}])
>
> db.catalog.insert([doc1, doc2])
BulkWriteResult({
  "writeErrors": [ ],
  "writeConcernErrors": [ ],
  "nInserted": 2,
  "nUpserted": 0,
  "nMatched": 0,
  "nModified": 0,
  "nRemoved": 0,
  "upserted": [ ]
})
>
```

Figure 2-27. Adding a Batch of Documents

MongoDB 2.6 Mongo shell added support for two new options: `writeConcern` and `ordered`. Next, we shall demonstrate the use of these options. The `writeConcern` option document supports the options listed in Table 2-3.

Table 2-3. Insert Method Options

Option	Description
w	Specifies the write concern.
j	Confirms that MongoDB has written the data to the on-disk journal. Boolean value, default value is false. MongoDB shouldn't be running with <code>-nojournal</code> option.
wtimeout	Applicable for w value greater than 1 specifies the time limit in milliseconds for the write concern. Write concern is the guarantee MongoDB provides when on reporting the success of an operation.

The write concern `w` may be set to the values listed in Table 2-4.

Table 2-4. Write Concern Values

w Value	Description
1	Provides acknowledgment of write concern on a stand-alone MongoDB or the primary in a replica set. Default value.
0	Disables acknowledgment of write operations.
majority	Confirms write operations to the majority of the replica set members.
Number greater than 1	Confirms write operations to the specified number of the replica set members.
tag set	Specifies in a fine-grained manner which replica set members must acknowledge the write operation.

Next, we shall add a batch of documents using the `writeConcern` and `ordered` options. Specify three BSON documents to be added.

```
doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
doc2 = {"_id" : ObjectId("53fb4b08d17e68cd481295d5"), "catalogId" : 1, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
doc3 = {"_id" : ObjectId("53fb4b08d17e68cd481295d6"), "catalogId" : 3, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}
```

Using the `db.collection.insert()` method and a `writeConcern` (with `w` set to `majority` and `wtimeout` set to 5000) and the `ordered` option set to true add the array of documents. Again, before running the `db.catalog.insert()` command drop the catalog collection.

```
>db.catalog.drop()
>db.catalog.insert([doc3, doc1, doc2], { writeConcern: { w: "majority", wtimeout: 5000 }, ordered:true })
```

A BulkWriteResult object gets returned with the nInserted value of 3 as shown in Figure 2-28.

```

> db.catalog.drop()
true
> doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId": 2, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}
<
    "_id": ObjectId("507f191e810c19729de860ea"),
    "catalogId": 2,
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Engineering as a Service",
    "author": "David A. Kelly"
>
>
> doc2 = {"_id": ObjectId("53fb4b08d17e68cd481295d5"), "catalogId": 1, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}
<
    "_id": ObjectId("53fb4b08d17e68cd481295d5"),
    "catalogId": 1,
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Quintessential and Collaborative",
    "author": "Tom Haunert"
>
>
> doc3 = {"_id": ObjectId("53fb4b08d17e68cd481295d6"), "catalogId": 3, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013"}
<
    "_id": ObjectId("53fb4b08d17e68cd481295d6"),
    "catalogId": 3,
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
>
>
> db.catalog.insert([doc3, doc1, doc2], { writeConcern: { w: "majority", wtimeout: 5000 }, ordered:true })
BulkWriteResult<{
    "writeErrors": [],
    "writeConcernErrors": [],
    "nInserted": 3,
    "nUpserted": 0,
    "nMatched": 0,
    "nModified": 0,
    "nRemoved": 0,
    "upserted": []
}>
>

```

Figure 2-28. Using the writeConcern Option

Subsequently invoke db.catalog.find() to list the three documents added. The documents are listed in the order specified in the document array added as shown in Figure 2-29.

```
>>> db.catalog.find()
[{"_id": ObjectId("53fb4b08d17e68cd481295d6"), "catalogId": 3, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013"}, {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId": 2, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("53fb4b08d17e68cd481295d5"), "catalogId": 1, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}]
```

Figure 2-29. Listing Documents with `find()`

A nonmajority `w` value of 1 cannot be used when a host is not a member of a replica set. For example, run the following commands.

```
>db.catalog.drop()
>db.catalog.insert([doc3, doc1, doc2], { writeConcern: { w: "1", wtimeout: 5000 },
ordered:true })
```

As indicated by the error message, `w` cannot be 1 as shown in Figure 2-30.

```
> db.catalog.drop()
true
> use catalog
switched to db catalog
>
> db.catalog.insert([doc3, doc1, doc2], { writeConcern: { w: "1", wtimeout: 5000 },
ordered:true })
WriteCommandError{
    "ok" : 0,
    "code" : 2,
    "errmsg" : "cannot use non-majority 'w' mode 1 when a host is not a member of a replica set"
}
```

Figure 2-30. Error Message “cannot use non-majority”

Saving a Document

The `db.collection.save()` method is used to save a document, which is different from the `db.collection.insert()` method as the insert method always adds a new document or throws an error if a document with the same `_id` field is already in the database. In contrast the `db.collection.save()` method updates the document if a document with the same `_id` already exists in the database and adds a new document if a document with the same `_id` does not already exist. When a document with the same `_id` already exists in the database the save method completely replaces the document with the new document. The `db.collection.save()` method was revised in the MongoDB 2.6 version. The following table, Table 2-5, lists the MongoDB 2.4 and MongoDB 2.6 (and later) version of the `save()` method.

Table 2-5. The Save Method

MongoDB Version	Save Method	Description
2.4	<code>db.collection.save(document)</code>	Saves a document.
2.6 and later	<code>db.collection.save(<document>, {writeConcern: <document>})</code>	Saves a document to the collection. The writeConcern is a new method argument.

In this section we shall demonstrate the different uses of the `save()` method. First, we shall update a document using the `save()` method. Add the following document using the `db.collection.insert()` method on the catalog collection.

1. Before adding the document drop the catalog collection and set the current database to catalog.

```
>db.catalog.drop()
>use catalog
>doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId"
: "catalog1", "journal" : 'Oracle Magazine', "publisher" : 'Oracle
Publishing', "edition" : 'November December 2013', "title" :
'Engineering as a Service', "author" : 'David A. Kelly'}
>db.catalog.insert(doc1)
```

2. The `db.collection.insert()` method adds the document and returns a `WriteResult` object with `nInserted` field as 1. Invoke the `db.collection.find()` method to list the document added.

```
>db.catalog.find()
```

3. Subsequently construct a document with the same `_id` as the added document but with some of the fields different.

```
>doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId"
: "catalog1", "journal" : 'Oracle Magazine', "publisher" : 'Oracle
Publishing', "edition" : '11-12-2013', "title" : 'Engineering as a
Service', "author" : 'Kelly, David A.'}
```

4. Invoke the `db.collection.save()` method to save the document with some of the fields modified. The `writeConcern` option, if specified, is ignored.

```
>db.catalog.save(doc1, { writeConcern: { w: "majority", wtimeout: 5000 } })
```

5. The `db.collection.save()` method saves the document and returns a `WriteResult` object with `nMatched` and `nModified` field value as 1. Because the same `_id` field value is used the document gets updated or modified with the new document. Subsequently invoke the `db.collection.find()` method on the catalog collection to list the document added and subsequently saved.

```
>db.catalog.find()
```

The document added with `db.collection.insert()` and subsequently saved with `db.collection.save()` gets listed. The edition field and the author field get modified in the saved (updated) document as shown in Figure 2-31.

```

> doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog1",
  "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
  'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A.
  Kelly'}
{
  "_id" : ObjectId("507f191e810c19729de860ea"),
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
}
>
> db.catalog.insert(doc1)
WriteResult({ "nInserted" : 1 })
>
> db.catalog.find()
{
  "_id" : ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog1", "journ
  al" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "Novembe
  r December 2013", "title" : "Engineering as a Service", "author" : "David A. Kel
  ly" }
>
> doc1 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 'catalog1',
  "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
  '11-12-2013', "title" : 'Engineering as a Service', "author" : 'Kelly, David A.'}
{
  "_id" : ObjectId("507f191e810c19729de860ea"),
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "11-12-2013",
  "title" : "Engineering as a Service",
  "author" : "Kelly, David A."
}
>
> db.catalog.save(doc1, { writeConcern: { w: "majority", wtimeout: 5000 } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.catalog.find()
{
  "_id" : ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog1", "journ
  al" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2
  013", "title" : "Engineering as a Service", "author" : "Kelly, David A." }
>

```

Figure 2-31. Updating a Document with `save()` Method

In the preceding example the document saved with the `save()` method has all of the same fields as the document added with the `insert()` method. In the next example we shall add new fields in the document saved with the `save` method rather than in the document added with the `insert` method.

1. Construct a BSON document with the `_id` field, the `catalogId` field, the `journal` field, the `publisher` field, and the `edition` field but without the `title` and `author` fields.

```

doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" :
  "catalog2", "journal" : 'Oracle Magazine', "publisher" : 'Oracle
  Publishing', "edition" : 'November December 2013'}

```

2. Add the document with the `db.collection.insert()` method in the `catalog` collection. Before running the command, drop the `catalog` collection.

```
>db.catalog.drop()
>db.catalog.insert(doc2)
```

3. The `insert()` method returns a `WriteResult` object with `nInserted` field value as 1. A subsequent invocation of the `db.collection.find()` method on the `catalog` collection lists the document.

```
>db.catalog.find()
```

4. Having added a document, next we shall update the document using the `save()` method. Construct a document with all of the same fields and field values as the document added with the `insert()` method including the `_id` field and additional fields of `title` and `author`.

```
>doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : '11-12-2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
```

5. Save the document with the `db.collection.save()` method in the `catalog` collection. The `catalog` collection is not to be dropped before the following command as we are saving (and modifying) the same document as added before.

```
>db.catalog.save(doc2)
```

6. The `save()` method modifies the document adding the fields `title` and `author`. The `save()` method returns a `WriteResult` object with the `nInserted` and `nModified` field values as 1. Subsequently invoke the `db.collection.find()` method to list the document.

```
>db.catalog.find()
```

The modified document gets listed with the two additional fields as shown in Figure 2-32. The document has been modified.

The screenshot shows a Windows command prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongo". The mongo shell is running, and the following commands and their results are displayed:

```

> db.catalog.drop()
true
>
> use catalog
switched to db catalog
>
> doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog2",
"journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
'November December 2013'}
{
    "_id" : ObjectId("507f191e810c19729de860ea"),
    "catalogId" : "catalog2",
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013"
}
>
> db.catalog.insert(doc2)
WriteResult({ "nInserted" : 1 })
>
> db.catalog.find()
[{"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013"}]
>
> doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : 'catalog2',
"journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : '11-12-2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
{
    "_id" : ObjectId("507f191e810c19729de860ea"),
    "catalogId" : "catalog2",
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "11-12-2013",
    "title" : "Quintessential and Collaborative",
    "author" : "Tom Haunert"
}
>
> db.catalog.save(doc2)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.catalog.find()
[{"_id": ObjectId("507f191e810c19729de860ea"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert"}]
>

```

Figure 2-32. Updating a Document and Adding New Fields with `save()` Method

In all of the preceding examples of `save()` we specified the `_id` field in the document added with `insert()` and subsequently saved with `save` method. In the next example of using the `save()` method we shall invoke the `save()` method with a different `_id` field value than in the document added with `insert()` method. Construct a BSON document as before and add the document using the `insert()` method. Drop the `catalog` collection before running the example.

1. A subsequent invocation of the `db.collection.find()` method lists the document added.

```
>db.catalog.drop()  
>doc2 = {"_id": ObjectId("507f191e810c19729de860ea"),  
  "catalogId" : "catalog2", "journal" : 'Oracle Magazine',  
  "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}  
> db.catalog.insert(doc2)  
>db.catalog.find()
```

2. Next, construct another document with the same fields and field values except the `_id` field value being different. Save the document with the `save` method.

```
>doc2 = {"_id": ObjectId("507f191e810c19729de860eb"), "catalogId"  
  : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle  
 Publishing', "edition" : '11-12-2013', "title" : 'Quintessential and  
 Collaborative', "author" : 'Tom Haunert'}  
>db.catalog.save(doc2)
```

3. Subsequently invoke the `db.collection.find()` method on the `catalog` collection.

```
db.catalog.find()
```

Because the document saved with the `save()` method has a different `_id` field value, a new document gets added and two documents get listed with `find()` as shown in Figure 2-33.

```

> db.catalog.drop()
true
> use catalog
switched to db catalog
> doc2 = { "_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013" }
{
    "_id": ObjectId("507f191e810c19729de860ea"),
    "catalogId": "catalog2",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
}
>
> db.catalog.insert(doc2)
WriteResult({ "nInserted": 1 })
>
> db.catalog.find()
{ "_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013" }
>
> doc2 = { "_id": ObjectId("507f191e810c19729de860eb"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "11-12-2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert" }
{
    "_id": ObjectId("507f191e810c19729de860eb"),
    "catalogId": "catalog2",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "11-12-2013",
    "title": "Quintessential and Collaborative",
    "author": "Tom Haunert"
}
>
> db.catalog.save(doc2)
WriteResult({
    "nMatched": 0,
    "nUpserted": 1,
    "nModified": 0,
    "_id": ObjectId("507f191e810c19729de860eb")
})
>
> db.catalog.find()
{ "_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013" }
{ "_id": ObjectId("507f191e810c19729de860eb"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "11-12-2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert" }
>

```

Figure 2-33. Upserting a Document with `save()`

The `nUpserted` in the `WriteResult` is 1. *Upsert* is just a term for an insert when no document matches a query document that could possibly match and update a document. *Upsert* is used in the context of operations that modify or update a document if a matching document is found. The `insert()` method does not modify or update a document and the term *upsert* cannot be used in the context of the `insert()` method. The `save()` and `update()` methods do modify or update a document and *upsert* is used in the context of these methods.

When a new `_id` field is used in the document added with `insert()` and subsequently saved with the `save()` method the process is called *upserting* the document.

When the `_id` field is not specified in the document with `insert()` and subsequently saved with `save()` an `insert` is performed instead of an `upsert`. Next, we shall discuss an example of `insert` using the `save()` method.

1. Construct a document as before but do not include an `_id` field. Add the document with the `insert()` method. A subsequent invocation of the `find()` method lists the document. Drop the `catalog` collection as before.

```
>db.catalog.drop()
>doc2 = {"catalogId" : "catalog2", "journal" : 'Oracle Magazine',
  "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}
>db.catalog.insert(doc2)
>db.catalog.find()
```

2. Next, construct another document with the `edition` field modified and two new fields `title` and `author`.

```
>doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine',
  "publisher" : 'Oracle Publishing', "edition" : '11-12-2013',
  "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
```

3. Invoke the `save()` method on the updated document.

```
>db.catalog.save(doc2)
```

4. A subsequent invocation of the `find()` method does not list an updated document but lists two documents.

```
>db.catalog.find()
```

Because the document added with `insert()` and the document saved with `save()` do not include the `_id` field, a new document gets added and gets listed with `find()` as shown in Figure 2-34. The `WriteResult` object returned has an `nInserted` field with a value as 1, indicating that a document has been added.

```

> use catalog
switched to db catalog
>
> doc2 = {"catalogId" : "catalog2", "journal" : 'Oracle Magazine', "publisher" :
'Oracle Publishing', "edition" : 'November December 2013'}
{
    "catalogId" : "catalog2",
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013"
}
>
> db.catalog.insert(doc2)
WriteResult({ "nInserted" : 1 })
>
> db.catalog.find()
[{"_id" : ObjectId("55ba5850403cc01aa5b078dd"), "catalogId" : "catalog2", "journal" :
'Oracle Magazine', "publisher" : "Oracle Publishing", "edition" : 'November December 2013'}
]
>
> doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" :
'Oracle Publishing', "edition" : '11-12-2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
{
    "catalogId" : "catalog2",
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "11-12-2013",
    "title" : "Quintessential and Collaborative",
    "author" : "Tom Haunert"
}
>
> db.catalog.save(doc2)
WriteResult({ "nInserted" : 1 })
>
> db.catalog.find()
[{"_id" : ObjectId("55ba5850403cc01aa5b078dd"), "catalogId" : "catalog2", "journal" :
'Oracle Magazine', "publisher" : "Oracle Publishing", "edition" : 'November December 2013'}
, {"_id" : ObjectId("55ba5850403cc01aa5b078de"), "catalogId" : "catalog2", "journal" :
'Oracle Magazine', "publisher" : "Oracle Publishing", "edition" : '11-12-2013', "title" :
'Quintessential and Collaborative', "author" : "Tom Haunert"}]
>

```

Figure 2-34. Inserting a Document with `save()`

Updating a Document

In this section we shall update a document. The `db.collection.update()` JavaScript method is used to update a document. The method has been revised in MongoDB 2.2 and 2.6. The different versions of the method are shown in Table 2-6.

Table 2-6. Different Versions of the update() Method

Version	Method	Description
Pre 2.2	<code>db.collection.update(query, update, <upsert>, <multi>)</code>	The <code><upsert></code> and <code><multi></code> are positional Boolean options.
2.2	<code>db.collection.update(query, update, options)</code> or <code>db.collection.update(</code> <code><query>,</code> <code><update>,</code> <code>{ upsert: <boolean>, multi: <boolean> }</code> <code>)</code>	Added the options parameter of type document for the <code><upsert></code> and <code><multi></code> options.
2.6 and later	<code>db.collection.update(query, update, options)</code> or <code>db.collection.update(</code> <code><query>,</code> <code><update>,</code> <code>{</code> <code>upsert: <boolean>,</code> <code>multi: <boolean>,</code> <code>writeConcern: <document></code> <code>}</code> <code>)</code>	The options parameter added the <code>writeConcern</code> option. Also the method returns a <code>WriteResult</code> with the status of the operation.

The `update()` method parameters including the options, which are optional, are discussed in Table 2-7.

Table 2-7. Update Method Parameters

Option	Type	Description
<code>query</code>	document	The query to select the document/s to update.
<code>update</code>	document	The update or modifications to apply.
<code>upsert</code>	Boolean	If set to true inserts a new document if a document by the selection criteria is not found. Default is false.
<code>multi</code>	Boolean	If set to true updates multiple documents if multiple documents are found with the selection criteria.
<code>writeConcern</code>	document	Specifies the write concern. Write concern is discussed earlier in this chapter.

Next, we shall update a document. First, we shall add a document using the `insert()` method and subsequently we shall update the added document using the `update` method.

1. In the mongo command shell run the following commands to add a document.
Before running the command, drop the catalog collection.

```
>db.catalog.drop()  
>doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',  
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}  
>db.catalog.insert(doc1)
```

2. Subsequently run the following command to find the added document.

```
>db.catalog.find()
```

3. To update the document invoke the `update()` method with a modified value for the `edition` field and new fields `title` and `author` fields. Include the `options` parameter consisting of the `upsert` option set to `true`.

```
db.catalog.update(  
  { catalogId: "catalog1" },  
  {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',  
  "publisher" : 'Oracle Publishing', "edition" : '11-12-2013',  
  "title" : 'Engineering as a Service', "author" : 'Kelly, David A.'},  
  { upsert: true}  
)
```

4. Subsequently invoke the `find()` method again.

```
>db.catalog.find()
```

The document gets updated and a `WriteResult` object is returned. The `nMatched` field in the `WriteResult` object has the value 1, `nUpserted` 0 and `nModified` 1 as shown in Figure 2-35.

```

> db.catalog.drop()
true
>
> use catalog
switched to db catalog
> doc1 = {"catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013"}
{
    "catalogId": "catalog1",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
}
>
>
> db.catalog.insert(doc1)
WriteResult<< "nInserted" : 1 >>
>
> db.catalog.find()
{ "_id" : ObjectId("55ba591f403cc01aa5b078df"), "catalogId" : "catalog1", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013" }
>
> db.catalog.update(
...   { catalogId: "catalog1" },
...   { "catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "11-12-2013", "title": "Engineering as a Service", "author": "Kelly, David A." },
...   { upsert: true }
... )
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> -

```

Figure 2-35. Updating a Document with update() Method

If key:value expressions are used in the update parameter the complete document is replaced with the document in the update parameter. If update operators expressions such as \$set are used in the update() method only the specified fields in the update operator expressions are replaced, not the complete document. If the update parameter contains update operators expressions it must not contain field key:value expressions.

Updating Multiple Documents

The update() method updates a single document by default. To update multiple documents use the multi parameter. A requirement for the update() method to update multiple documents is to use the update operators expressions. If the update() method specifies only field:value expressions it cannot update multiple documents.

Next, we shall update multiple documents in MongoDB 3.0.5.

1. First delete any previously added catalog collection by invoking db.catalog.drop(). Add two documents to the catalog collection using the db.collection.insert() method. Subsequently list the documents in the catalog collection using the db.collection.find() method.

```

> db.catalog.drop()
>use catalog
>doc1 = {"catalogId": 1, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}

```

```

db.catalog.insert(doc1)
>doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine', "publisher"
: 'Oracle Publishing', "edition" : 'November December 2013','title" :
'Quintessential and Collaborative',"author" : 'Tom Haunert'}
>db.catalog.insert(doc2)
>db.catalog.find()

```

2. Next, invoke the db.collection.update() method using update operators expressions with \$set updating the edition field and \$inc updating the catalogId field. In the options parameter specify multi option as true and also specify the writeConcern option. The wtimeout though specified for w:1 is applicable only for w>1 and is ignored for w:1. Subsequently invoke the db.catalog.find() method to list the updated documents.

```

>db.catalog.update(
  { journal: 'Oracle Magazine'},
  {
    $set: { edition: '11-12-2013' },
    $inc: { catalogId: 2 }
  },
  {
    multi: true,
    writeConcern: { w: 1, wtimeout: 5000 }
  }
)
>db.catalog.find()

```

As the output from the update() method indicates, two documents get matched and two documents get updated as shown in Figure 2-36.

```

Administrator: C:\Windows\system32\cmd.exe - mongo
> doc1 = {"catalogId": 1, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}
{
    "catalogId": 1,
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Engineering as a Service",
    "author": "David A. Kelly"
}
> db.catalog.insert(doc1)
WriteResult({ "nInserted": 1 })
> doc2 = {"catalogId": 2, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}
{
    "catalogId": 2,
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Quintessential and Collaborative",
    "author": "Tom Haunert"
}
> db.catalog.insert(doc2)
WriteResult({ "nInserted": 1 })
> db.catalog.find()
{
  "_id": ObjectId("55ba5994403cc01aa5b078e0"),
  "catalogId": 1,
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "November December 2013",
  "title": "Engineering as a Service",
  "author": "David A. Kelly"
}
{
  "_id": ObjectId("55ba5995403cc01aa5b078e1"),
  "catalogId": 2,
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "November December 2013",
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}
>
> db.catalog.update(
...   { journal: 'Oracle Magazine' },
...   {
...     $set: { edition: '11-12-2013' },
...     $inc: { catalogId: 2 }
...   },
...   {
...     multi: true,
...     writeConcern: { w: 1, wtimeout: 5000 }
...   }
... )
WriteResult({ "nMatched": 2, "nUpserted": 0, "nModified": 2 })
> db.catalog.find()
{
  "_id": ObjectId("55ba5994403cc01aa5b078e0"),
  "catalogId": 3,
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "11-12-2013",
  "title": "Engineering as a Service",
  "author": "David A. Kelly"
}
{
  "_id": ObjectId("55ba5995403cc01aa5b078e1"),
  "catalogId": 4,
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "11-12-2013",
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}
>

```

Figure 2-36. Updating Multiple Documents with update() Method

Finding One Document

The `findOne()` method may be used to find a single document from a collection. The syntax of the `findOne()` method is as follows.

```
db.collection.findOne(query, <projection>)
```

Both the method parameters are optional. The `query` parameter specifies the query selection criteria and is of type document. The `<projection>` parameter specifies the fields to return. The type of the `query` and `<projection>` parameters is document. By default all fields are returned. If multiple documents match a selection criteria or if no selection criteria is specified and the collection has multiple documents the first document that is matched is returned.

Next, we shall add two documents and find one document using `findOne()` method without any args.

1. Drop the catalog collection and add two documents to the catalog collection using the `db.collection.insert()` method.

```
>db.catalog.drop()
>doc1 = {"catalogId" : "catalog1", "journal" : 'Oracle Magazine',
  "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
  "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
>db.catalog.insert(doc1)
>doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId"
  :"catalog2", "journal" : 'Oracle Magazine', "publisher" : 'Oracle
  Publishing', "edition" : 'November December 2013'}
>db.catalog.insert(doc2)
```

2. Find one document using the `findOne()` method.

```
>db.catalog.findOne()
```

One of the two documents gets returned as shown in Figure 2-37.



The screenshot shows a Windows command prompt window titled "Administrator: C:\Windows\system32\cmd.exe - mongo". The MongoDB shell is running. The user performs the following steps:

- Creates a new database named "catalog" with the command `db.catalog.drop()`.
- Selects the "catalog" database with the command `use catalog`.
- Creates two documents, `doc1` and `doc2`, with specific fields like catalogId, journal, publisher, edition, title, and author.
- Inserts `doc1` into the collection with the command `db.catalog.insert(doc1)`.
- Inserts `doc2` into the collection with the command `db.catalog.insert(doc2)`.
- Finds one document in the collection with the command `db.catalog.findOne()`.
- Exits the MongoDB shell with the command `exit`.

```

> db.catalog.drop()
true
> use catalog
switched to db catalog
>
> doc1 = {"catalogId": "catalog1", "journal": 'Oracle Magazine', "publisher": 'Oracle Publishing', "edition": 'November December 2013', "title": 'Engineering as a Service', "author": 'David A. Kelly'}
{
    "catalogId": "catalog1",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Engineering as a Service",
    "author": "David A. Kelly"
}
>
> db.catalog.insert(doc1)
WriteResult({ "nInserted": 1 })
>
> doc2 = {"_id": ObjectId("507f191e810c19729de860ea"), "catalogId": "catalog2", "journal": 'Oracle Magazine', "publisher": 'Oracle Publishing', "edition": 'November December 2013'}
{
    "_id": ObjectId("507f191e810c19729de860ea"),
    "catalogId": "catalog2",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
}
>
> db.catalog.insert(doc2)
WriteResult({ "nInserted": 1 })
>
> db.catalog.findOne()
{
    "_id": ObjectId("55ba5a1c403cc01aa5b078e2"),
    "catalogId": "catalog1",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013",
    "title": "Engineering as a Service",
    "author": "David A. Kelly"
}
> -

```

Figure 2-37. Finding One Document

Finding All Documents

To find all documents the `find()` method must be used. The `find()` method has the following syntax.

```
db.collection.find(query, <projection>)
```

The `query` parameter specifies the query selection criteria and the `<projection>` parameter specifies the fields to return. None of the parameters is required and both are of type document. The `find()` method actually returns a cursor and if the method is invoked in the MongoDB shell the cursor is automatically iterated to display the first 20 documents. To run the iterator on the remaining document specify it in the shell.

To find all documents in the `catalog` collection that may have been added previously and not removed, run the `find()` method.

```
>db.catalog.find()
```

All the documents in the catalog collection get returned as shown in Figure 2-38. To distinguish between the single document returned by the `findOne()` method and all documents returned by `find()`, the results from both `findOne()` and `find()` are listed.

```

> db.catalog.findOne()
{
  "_id" : ObjectId("55ba5a1c403cc01aa5b078e2"),
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
}
> db.catalog.find()
[ {
  "_id" : ObjectId("55ba5a1c403cc01aa5b078e2"),
  "catalogId" : "catalog1",
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
},
  {
    "_id" : ObjectId("507f191e810c19729de860ea"),
    "catalogId" : "catalog2",
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013"
  }
]
>

```

Figure 2-38. Finding All Documents with `find()` Method

Finding Selected Fields

Both the `find()` and `findOne()` methods support the `<projection>` parameter as mentioned before. The `<projection>` parameter specifies the fields to return with a document of the following syntax.

```
{ field1: <boolean>, field2: <boolean> ... }
```

To include a field set the boolean to true or 1. To exclude a field set the field to false or 0. For example, add two documents to the catalog collection using the `insert()` method. But, first drop the catalog collection.

```

>db.catalog.drop()
>doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing',
"edition" : 'November December 2013', "title" : 'Engineering as a Service',
"author" : 'David A. Kelly'}
>db.catalog.insert(doc1)
>doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing',
"edition" : 'November December 2013', "title" : 'Quintessential and Collaborative',
"author" : 'Tom Haunert'}
>db.catalog.insert(doc2)

```

Subsequently invoke the `findOne()` method with the query parameter set to an empty document and the `<projection>` parameter set to include the edition, title, and author fields.

```

>db.catalog.findOne(
  { },
  { edition: 1, title: 1, author: 1 }
)

```

A single document gets returned and only the title, author, and edition fields get returned as shown in Figure 2-39. The _id field is always returned and is not required to be specified in the <projection> parameter as one of the fields to return.

```

> db.catalog.drop()
true
> doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
{
    "catalogId" : 1,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013",
    "title" : "Engineering as a Service",
    "author" : "David A. Kelly"
}
>
>
> db.catalog.insert(doc1)
WriteResult({ "nInserted" : 1 })
>
> doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
{
    "catalogId" : 2,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013",
    "title" : "Quintessential and Collaborative",
    "author" : "Tom Haunert"
}
>
>
> db.catalog.insert(doc2)
WriteResult({ "nInserted" : 1 })
>
> db.catalog.findOne(
...     { },
...     { edition: 1, title: 1, author: 1 }
... )
{
    "_id" : ObjectId("55ba5ae4403cc01aa5b078e3"),
    "edition" : "November December 2013",
    "title" : "Engineering as a Service",
    "author" : "David A. Kelly"
}
>

```

Figure 2-39. Finding Selected Fields

The <projection> parameter may specify to either include field/s or exclude field/s, not both. To demonstrate invoke the findOne() method with edition field excluded, and title and author fields included.

```
>db.catalog.findOne(
    { },
    { edition: 0, title: true, author: 1 }
)
```

An exception gets generated indicating that including and excluding fields cannot be mixed as shown in Figure 2-40.



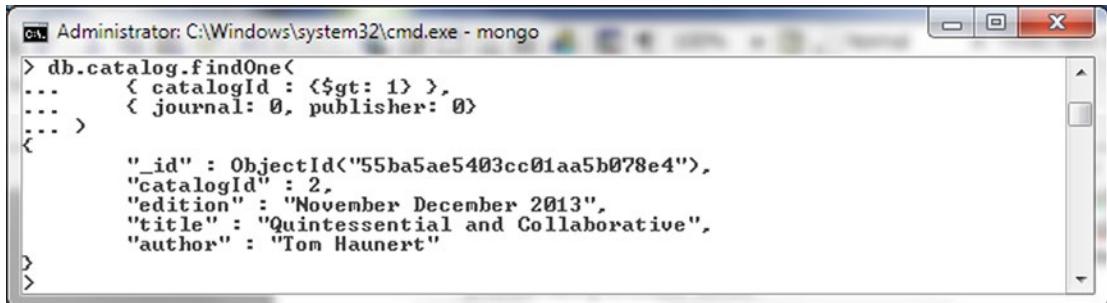
```
> db.catalog.findOne(
...   { },
...   { edition: 0, title: true, author: 1 }
...
2015-07-30T10:13:24.357-0700 E QUERY    Error: error: <
"err": "Can't canonicalize query: BadValue Projection cannot have a mi
x of inclusion and exclusion.",
"code": 17287
>
at Error (<anonymous>)
at DBQuery.next (src/mongo/shell/query.js:259:15)
at DBCollection.findOne (src/mongo/shell/collection.js:189:22)
at <shell>:1:12 at src/mongo/shell/query.js:259
>
```

Figure 2-40. Including and Excluding Fields Cannot Be Mixed

To exclude fields all fields in the `<projection>` argument must be set to be excluded. For example, in the following invocation of `findOne()` the `catalogId` field is set using the query comparison operator `$gt` and the `<projection>` parameter value is set to exclude the `journal` and `publisher` fields.

```
>db.catalog.findOne(
  { catalogId : {$gt: 1} },
  { journal: 0, publisher: 0}
)
```

The `findOne()` method returns only the document with `catalogId` greater than 1 and excludes the `journal` and `publisher` fields as shown in Figure 2-41.



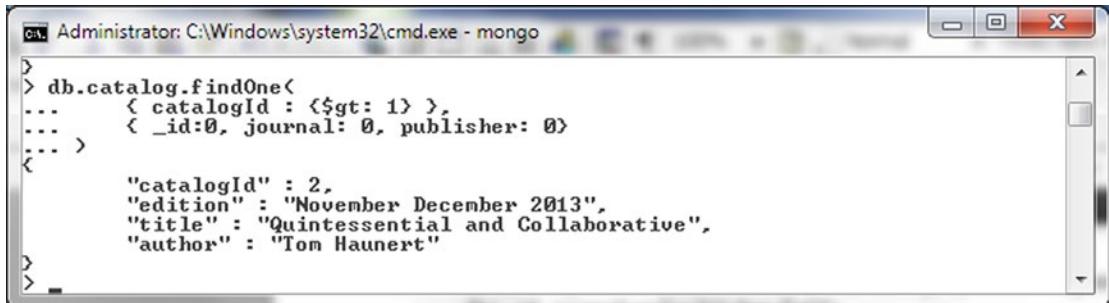
```
> db.catalog.findOne(
...   { catalogId : {$gt: 1} },
...   { journal: 0, publisher: 0 }
...
{
  "_id" : ObjectId("55ba5ae5403cc01aa5b078e4"),
  "catalogId" : 2,
  "edition" : "November December 2013",
  "title" : "Quintessential and Collaborative",
  "author" : "Tom Haunert"
}
```

Figure 2-41. Finding a Document with `findOne()` Method

Even the `_id` field may be excluded in the `<projection>` argument. For example, exclude the `_id`, `journal` and `publisher` fields.

```
>db.catalog.findOne(
  { catalogId : {$gt: 1} },
  { _id:0, journal: 0, publisher: 0}
)
```

The `_id` field is also excluded in the output as shown in Figure 2-42.



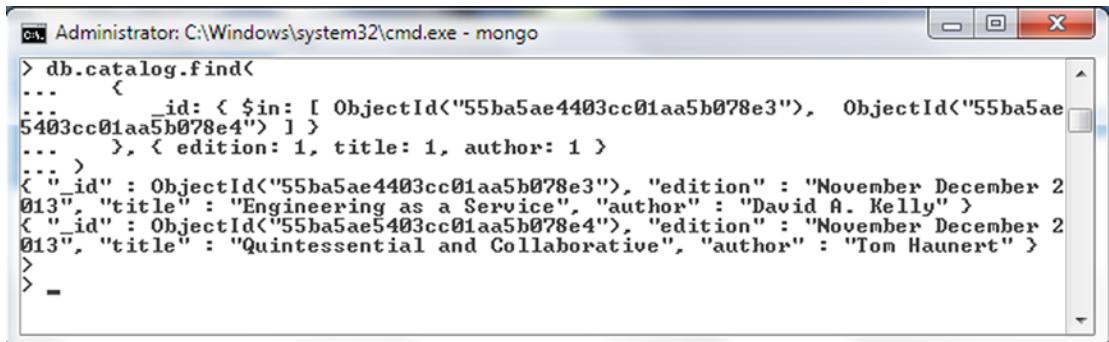
```
> db.catalog.findOne(
...   { catalogId : {$gt: 1},
...     _id:0, journal: 0, publisher: 0}
...
{
  "catalogId" : 2,
  "edition" : "November December 2013",
  "title" : "Quintessential and Collaborative",
  "author" : "Tom Haunert"
>
```

Figure 2-42. Excluding the `_id` Field

If the `_id` field value is to be specified in the `query` arg of the `find()` or `findOne()` method, the `ObjectId` wrapper class must be used. For example, specify the `query` arg using the comparison query operator `$in` and the `ObjectId` to specify the `_id` field values. The same `_id` field values were used in adding the documents. The `_id` field values would be different for different users; use the `_id` field values for documents added previously. The `_id` field values may be found using the `db.catalog.find()` command.

```
>db.catalog.find(
  {
    _id: { $in: [ ObjectId("55ba5ae4403cc01aa5b078e3"),
      ObjectId("55ba5ae5403cc01aa5b078e4") ] }
  }, { edition: 1, title: 1, author: 1 }
)
```

The `find()` method returns the two documents with the specified `_id` field values as shown in Figure 2-43. Only the fields specified in the `<projection>` arg are returned.



```
> db.catalog.find(
...   {
...     _id: { $in: [ ObjectId("55ba5ae4403cc01aa5b078e3"), ObjectId("55ba5ae5403cc01aa5b078e4") ] }
...   }, { edition: 1, title: 1, author: 1 }
...
< "_id" : ObjectId("55ba5ae4403cc01aa5b078e3"), "edition" : "November December 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" >
< "_id" : ObjectId("55ba5ae5403cc01aa5b078e4"), "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" >
>
> -
```

Figure 2-43. Using the Comparison Query Operator `$in` to Find Documents

Using the Cursor

As mentioned before the `db.collection.find()` method returns a cursor and when the `find()` method is invoked in the shell the first 20 documents are iterated over and displayed by default. The cursor supports several methods and a handle to the cursor object may be obtained to invoke the methods. Some of the methods supported by the cursor are discussed in Table 2-8.

Table 2-8. Methods Supported by Cursor

Method	Description
<code>batchSize()</code>	Specifies the number of documents returned in a single network message.
<code>count()</code>	Number of documents in the cursor.
<code>forEach(<function>)</code>	Iterates over each document to apply a JavaScript function.
<code>hasNext()</code>	Returns true if the cursor has another document to iterate over or returns false if the cursor does not.
<code>limit()</code>	Limits the size of the cursor's result set.
<code>next()</code>	Returns the next document in the cursor.
<code>skip()</code>	Specifies the number of documents to skip before getting documents from the MongoDB database.
<code>sort()</code>	Returns results in the specified sort order.
<code>toArray()</code>	Returns an array of documents.

Next, we shall demonstrate the use of some of the cursor methods. Remove the `catalog` collection and add three documents to the `catalog` collection using the `insert()` method.

```
>db.catalog.drop()
>doc1 = {"_id" : ObjectId("53fb4b08d17e68cd481295d5"), "catalogId" : 1, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
db.catalog.insert(doc1)
>doc2 = {"_id" : ObjectId("53fb4b08d17e68cd481295d6"), "catalogId" : 2, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
>db.catalog.insert(doc2)
>doc3 = {"_id" : ObjectId("53fb4b08d17e68cd481295d7"), "catalogId" : 3, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}
>db.catalog.insert(doc3)
```

Invoke the `forEach` method on the cursor returned and invoke the `printjson` function to output the document as JSON.

```
>db.catalog.find().forEach(printjson)
```

The three documents get displayed in JSON format as shown in Figure 2-44.

```

> db.catalog.find().forEach(printjson)
{
  "_id" : ObjectId("53fb4b08d17e68cd481295d5"),
  "catalogId" : 1,
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
}
{
  "_id" : ObjectId("53fb4b08d17e68cd481295d6"),
  "catalogId" : 2,
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Quintessential and Collaborative",
  "author" : "Tom Haunert"
}
{
  "_id" : ObjectId("53fb4b08d17e68cd481295d7"),
  "catalogId" : 3,
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013"
}
> -

```

Figure 2-44. Listing Document JSON with Cursor

In the previous example we did not create a variable for the cursor but invoked the `forEach()` method in sequence to the `find()` method invocation. We may also specify a variable for the cursor as follows. A variable may be used if the cursor is to be used for some other purpose also.

```
var cursor= db.catalog.find();
```

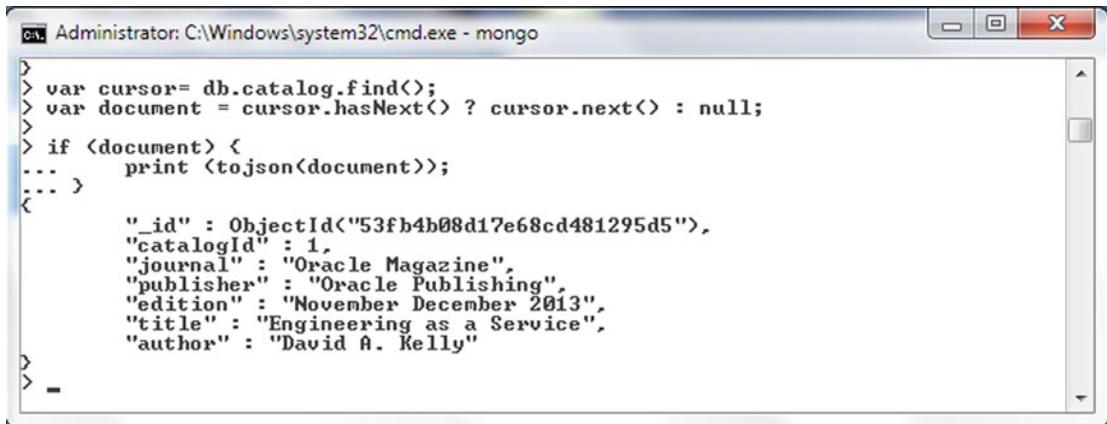
Subsequently the `hasNext()` and `next()` methods are invoked in a ternary/conditional operator to find the next document in the cursor.

```
var document = cursor.hasNext() ? cursor.next() : null;
```

If a document is returned, print the JSON form of the document.

```
if (document) {
  print (tojson(document));
}
```

The JSON form of a document gets output as shown in Figure 2-45.



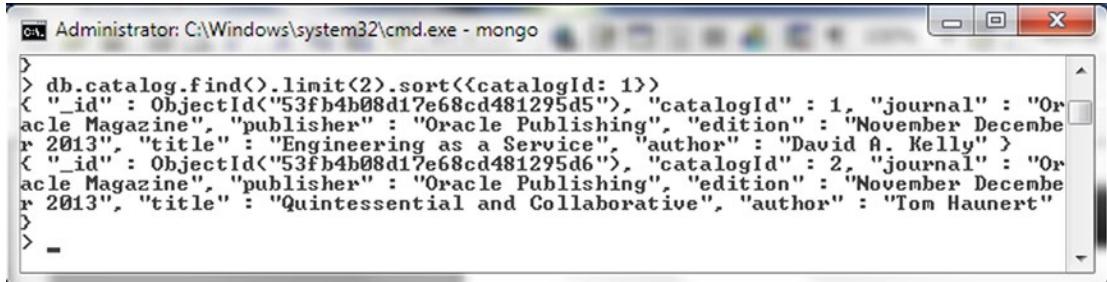
```
> var cursor = db.catalog.find();
> var document = cursor.hasNext() ? cursor.next() : null;
>
> if (document) {
...     print (toJson(document));
... }
{
    "_id" : ObjectId("53fb4b08d17e68cd481295d5"),
    "catalogId" : 1,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013",
    "title" : "Engineering as a Service",
    "author" : "David A. Kelly"
}
> -
```

Figure 2-45. Using a Variable for the Cursor

The cursor methods may be invoked in sequence as in the following `find()` method invocation in which the `limit()` method is invoked followed by the `sort()` method. The `limit()` method limits the documents returned to two and the `sort` method sorts by `catalogId` in ascending order.

```
>db.catalog.find().limit(2).sort({catalogId: 1})
```

Two documents with `catalogId` field in ascending order get returned as shown in Figure 2-46.



```
>
> db.catalog.find().limit(2).sort({catalogId: 1})
{
    "_id" : ObjectId("53fb4b08d17e68cd481295d5"),
    "catalogId" : 1,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013",
    "title" : "Engineering as a Service",
    "author" : "David A. Kelly"
}
{
    "_id" : ObjectId("53fb4b08d17e68cd481295d6"),
    "catalogId" : 2,
    "journal" : "Oracle Magazine",
    "publisher" : "Oracle Publishing",
    "edition" : "November December 2013",
    "title" : "Quintessential and Collaborative",
    "author" : "Tom Haunert"
}
> -
```

Figure 2-46. Invoking Cursor Methods in Sequence

Finding and Modifying a Document

The `findAndModify()` method may be used to find and modify a single document and has the following syntax.

```
db.collection.findAndModify({
    query: <document>,
    sort: <document>,
    remove: <boolean>,
    update: <document>,
    new: <boolean>,
    fields: <document>,
    upsert: <boolean>
})
```

The following parameters listed in Table 2-9 are supported by the `findAndModify()` method. All parameters are optional except that one of update or remove must be specified.

Table 2-9. Parameters Supported by `findAndModify()` Method

Parameter	Type	Description
query	document	The query selection criteria to find a document to modify. If multiple documents are returned by the selection criteria, only one of the documents is modified.
sort	document	Specifies which document is modified if multiple documents are returned. The first document in the sort order is specified by the <code>sort</code> parameter.
remove	boolean	Specifies if the selected document is to be removed. Default is false. Either <code>remove</code> or <code>upsert</code> must be specified.
update	document	Specifies the update to apply. The <code>update</code> parameter makes use of update operators or <code>field:value</code> .
new	boolean	Specifies whether the modified document is to be returned or the original. Default is false. If <code>remove</code> is set to true new is ignored.
fields	document	Subset of fields to return specified in the <projection> format <code>{field1:1, field2:1}</code> .
upsert	boolean	Specifies if a new document is to be created and returned if a document to match the selection criteria is not found. Default is false. Either <code>remove</code> or <code>upsert</code> must be specified.

For example, invoke the `findAndModify()` method as follows:

- Set the `query` parameter to find documents with the `journal` field as Oracle Magazine.
- Sort by `catalogId` field in ascending order.
- Specify the `update` parameter with update operators `$inc` to increment the `catalogId` field by 1 and the `$set` operator to set a new value for the `edition` field.
- The `upsert` parameter is set to `true` and so is the `new` parameter.
- The `fields` parameter specifies the fields to return as `catalogId`, `edition`, `title`, and `author`.

```
>db.catalog.findAndModify({
  query: {journal : "Oracle Magazine"},
  sort: {catalogId : 1},
  update: {$inc: {catalogId: 1}, $set: {edition: '11-12-2013'}},
  upsert :true,
  new: true,
  fields: {catalogId: 1, edition: 1, title: 1, author: 1}
})
```

Before invoking the preceding method add some documents, which should have the `catalogId` field value as a number because the `$inc` operator cannot be applied to a non-number. Because the parameter `new` is set to `true` the method returns the modified document as shown in Figure 2-47.

```

Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.findAndModify({
...   query: {journal : "Oracle Magazine"},
...   sort: {catalogId : 1},
...   update: {$inc: {catalogId: 1}, $set: {edition: '11-12-2013'}},
...   upsert :true,
...   new: true,
...   fields: {catalogId: 1, edition: 1, title: 1, author: 1}
... })
{
  "_id" : ObjectId("53fb4b08d17e68cd481295d5"),
  "catalogId" : 2,
  "edition" : "11-12-2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
}
>

```

Figure 2-47. Using the *findAndModify* Method

Both *upsert* and *remove* parameter values cannot be specified in the method invocation. For example, run the following command.

```

db.catalog.findAndModify({
  query: {journal : "Oracle Magazine"},
  sort: {catalogId : 1},
  update: {$inc: {catalogId: 1}, $set: {edition: '11-12-2013'}},
  remove: true,
  upsert :true,
  new: true,
  fields: {catalogId: 1, edition: 1, title: 1, author: 1}
})

```

As indicated by the following exception both *upsert* and *remove* parameter values cannot coexist as shown in Figure 2-48.

```

Administrator: C:\Windows\system32\cmd.exe - mongo
>
> db.catalog.findAndModify({
...   query: {journal : "Oracle Magazine"},
...   sort: {catalogId : 1},
...   update: {$inc: {catalogId: 1}, $set: {edition: '11-12-2013'}},
...   remove: true,
...   upsert :true,
...   new: true,
...   fields: {catalogId: 1, edition: 1, title: 1, author: 1}
... })
2015-08-01T17:43:13.698-0700 E QUERY      Error: findAndModifyFailed failed: { "ok" : 0, "errmsg" : "remove and upsert can't co-exist" }
at Error (<anonymous>)
at DBCollection.findAndModify (src/mongo/shell/collection.js:636:15)
at <shell>:1:12 at src/mongo/shell/collection.js:636
>

```

Figure 2-48. Error Message “remove and upsert can’t co-exist”

Removing a Document

The `db.collection.remove()` method is used to remove document/s. The method has the following syntax in MongoDB version prior to 2.6.

```
db.collection.remove(
  <query>,
  <justOne>
)
```

The `remove()` method has the following syntax in version 2.6 and later.

```
db.collection.remove(
  <query>,
  {
    justOne: <boolean>,
    writeConcern: <document>
  }
)
```

The method parameters are as listed in Table 2-10.

Table 2-10. The `remove` Method Parameters

Parameter	Type	Description
query	document	Specifies deletion criteria using query operators. To remove all documents specify an empty document {}. In pre-2.6 versions all documents may also be deleted by omitting the query.
justOne	boolean	Specifies if only a single document is to be removed. Default is false, which implies all documents are removed.
writeConcern	document	The write concern, which is discussed in an earlier section.

In version 2.6 and later the `remove()` method returns a `WriteResult` object. As an example add three documents, one with `catalogId` 3 and two with `catalogId` 2.

```
>db.catalog.drop()
>doc1 = {"_id" : ObjectId("53fb4b08d17e68cd481295d5"), "catalogId" : 2, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Engineering as a Service', "author" : 'David A. Kelly'}
d b.catalog.insert(doc1)
>doc2 = {"_id" : ObjectId("53fb4b08d17e68cd481295d6"), "catalogId" : 2, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'}
>db.catalog.insert(doc2)
>doc3 = {"_id" : ObjectId("53fb4b08d17e68cd481295d7"), "catalogId" : 3, "journal" :
'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}
>db.catalog.insert(doc3)
```

Subsequently remove the document/s with catalogId as 2 as follows.

```
>db.catalog.remove({ catalogId: 2 })
```

As indicated in the output in Figure 2-49 the two documents with catalogId as 2 are removed (nRemoved is 2) and only the document with catalogId as 3 is returned with the find() method invocation subsequent to removing the documents.

```
> db.catalog.remove({ catalogId: 2 })
WriteResult({ "nRemoved" : 2 })
> db.catalog.find()
{ "_id" : ObjectId("53fb4b08d17e68cd481295d7"), "catalogId" : 3, "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013" }
>
```

Figure 2-49. Removing Document/s with remove()

In another example, specify the query using the comparison query operator \$gt for the catalogId field. Set justOne to true to remove only one document and set the writeConcern to { w: 1, wtimeout: 5000 }. The value of 1 for the w option provides acknowledgment of write on a single MongoDB server. The wtimeout if specified for w:1 is ignored implying that the a timeout does not occur and an acknowledgment from a single server is returned.

```
>db.catalog.remove(
  { catalogId: { $gt: 1 } },
  {justOne:true, writeConcern: { w: 1, wtimeout: 5000 } }
)
```

The nRemoved field in the WriteResult object returned indicates that one document got removed as shown in Figure 2-50.

```
> db.catalog.remove(
...   { catalogId: { $gt: 1 } },
...   {justOne:true, writeConcern: { w: 1, wtimeout: 5000 } }
...
)
WriteResult({ "nRemoved" : 1 })
```

Figure 2-50. Using a Comparison Query Operator \$gt in remove() Method

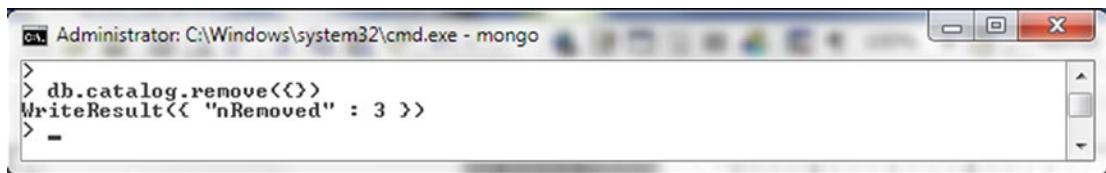
To remove all documents in pre-2.6 version the remove() method is invoked as follows without any args.

```
>db.catalog.remove()
```

To remove all documents from a MongoDB version 2.6 and later collection an empty document {} must be provided as an argument as follows.

```
>db.catalog.remove({})
```

The `remove()` method returns a `WriteResult` object with `nRemoved` as 3 indicating that three documents have been removed as shown in Figure 2-51.



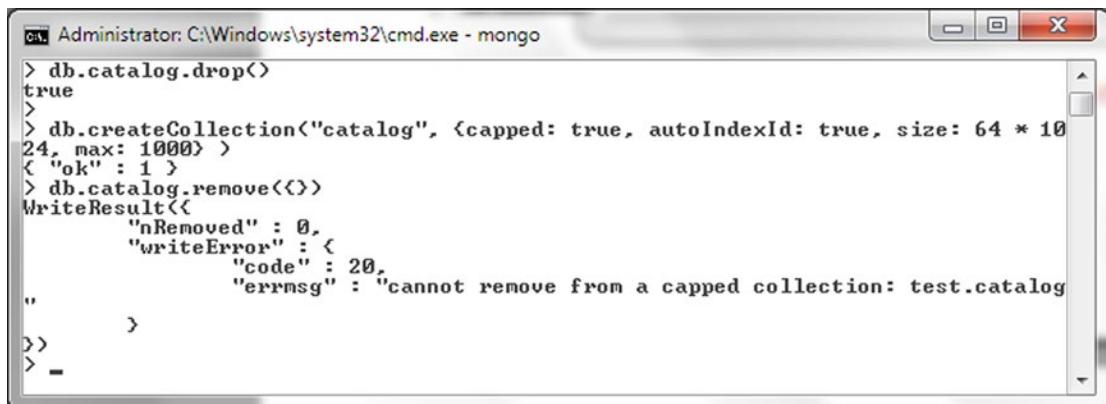
```
> db.catalog.remove(<>)
WriteResult<{ "nRemoved" : 3 }>
> -
```

Figure 2-51. Removing All Documents

A document cannot be removed from a capped collection, which is a fixed size collection similar to a circular buffer, using the `remove()` method. To demonstrate, invoke the `remove()` method on a capped collection; first drop the `catalog` collection and create a capped collection as discussed earlier in the section “Creating a Collection.”

```
>db.catalog.drop()
> db.createCollection("catalog", {capped: true, autoIndexId: true, size: 64 * 1024, max: 1000} )
>db.catalog.remove({})
```

An error message gets displayed as shown in Figure 2-52.



```
> db.catalog.drop()
true
> db.createCollection("catalog", {capped: true, autoIndexId: true, size: 64 * 1024, max: 1000} )
{ "ok" : 1 }
> db.catalog.remove(<>)
WriteResult<{
    "nRemoved" : 0,
    "writeError" : {
        "code" : 20,
        "errmsg" : "cannot remove from a capped collection: test.catalog"
    }
}
> -
```

Figure 2-52. A Capped Collection Cannot Be Deleted

Summary

In this chapter we discussed some of the salient Mongo shell methods and commands. In the next chapter we shall use PHP with MongoDB server.

CHAPTER 3



Using MongoDB with PHP

PHP continues to be one of the most commonly used scripting languages used in developing web sites. The PHP Driver for MongoDB may be used to connect to MongoDB, and create a collection and perform CRUD (Create, Read, Update, Delete) operations on the database. The PHP driver extension dll is not packaged with the PHP download but is required to be added and configured. In this chapter we shall use the PHP Driver for MongoDB to connect to MongoDB database server and add, find, update, and delete data from the server. This chapter covers the following topics:

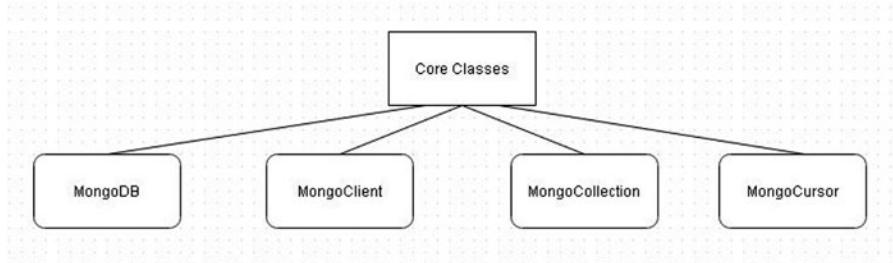
- Getting Started
- Using Collections
- Using Documents

Getting Started

In the following subsections we shall discuss the PHP MongoDB Database Driver. We also discuss setting up the environment and creating a connection to MongoDB Driver from a PHP script.

Overview of the PHP MongoDB Database Driver

The PHP MongoDB Driver provides several classes for connecting to MongoDB and performing CRUD operations. The core classes in the PHP driver are shown in Figure 3-1.

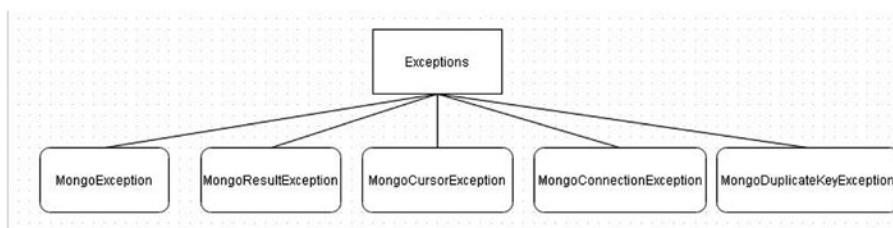
**Figure 3-1.** Core Classes in PHP MongoDB Driver

The main core classes are discussed in Table 3-1.

Table 3-1. PHP MongoDB Driver Core Classes

Class	Description
MongoDB	Instances of the class are used to interact with MongoDB database. The class constructor <code>MongoDB::__construct (MongoClient \$conn , string \$name)</code> creates a new database, but the constructor is not supposed to be called directly. Instead an instance of MongoDB is created using <code>MongoClient::__get()</code> or <code>MongoClient::selectDB()</code> method.
MongoClient	The class is used to create and manage connections with MongoDB. The class constructor <code>MongoClient::__construct ([string \$server = "mongodb://localhost:27017" [, array \$options = array("connect" => TRUE) [, array \$driver_options]]])</code> creates a new client connection.
MongoCollection	The class represents a MongoDB collection.
MongoCursor	The class represents a cursor that is used to iterate a result set of a database query.

MongoDB server operations may generate exceptions. Some of the main exceptions are illustrated in Figure 3-2.

**Figure 3-2.** Types of Exceptions Generated by MongoDB

The main exceptions are discussed in Table 3-2.

Table 3-2. The Main Exception Classes

Exception Class	Description
MongoException	The default exception class that all other exception classes must extend. Thrown by <code>MongoCollection::batchInsert</code> <code>MongoCollection::insert</code> methods if the inserted documents are empty.
MongoResultException	Several command helpers such as <code>MongoCollection::findAndModify</code> throw the exception.
MongoCursorException	Any database request that fails to receive an expected reply throws the exception. <code>MongoCollection::update</code> , <code>MongoCollection::batchInsert</code> and <code>MongoCollection::insert</code> throw the exception if the <code>w</code> option is set and the write fails.
MongoConnectionException	Thrown when the driver fails to connect to the database. <code>MongoClient::connect</code> and <code>MongoCollection::findOne</code> throw <code>MongoConnectionException</code> if they fail to connect to the database.
MongoDuplicateKeyException	Thrown if a document is inserted into a collection and the collection already has another document with the same unique key.

Setting Up the Environment

We need to install the following software in addition to installing MongoDB server.

- MongoDB Server
- PHP
- PHP Driver for MongoDB

Download and install MongoDB 3.0.5. Add the MongoDB bin directory, for example `C:\Program Files\MongoDB\Server\3.0\bin` directory to the PATH environment variable. Start the MongoDB server with the following command.

```
>mongod
```

The output from the command indicates that the MongoDB server has started as shown in Figure 3-3.

```
C:\MongoDB>mongod
2015-07-31T08:28:26.183-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-07-31T08:28:26.311-0700 I JOURNAL [initandlisten] journal dir=C:/data/db/j
ournal
2015-07-31T08:28:26.311-0700 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-07-31T08:28:26.347-0700 I JOURNAL [durability] Durability thread started
2015-07-31T08:28:26.355-0700 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-07-31T08:28:26.405-0700 I CONTROL [initandlisten] MongoDB starting : pid=2
452 port=27017 dbpath=C:/data/db/ 64-bit host=dvohra-PC
2015-07-31T08:28:26.405-0700 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-07-31T08:28:26.406-0700 I CONTROL [initandlisten] db version v3.0.5
2015-07-31T08:28:26.406-0700 I CONTROL [initandlisten] git version: 8hc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-07-31T08:28:26.406-0700 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-07-31T08:28:26.407-0700 I CONTROL [initandlisten] allocator: tcmalloc
2015-07-31T08:28:26.408-0700 I CONTROL [initandlisten] options: <>
2015-07-31T08:28:28.558-0700 I NETWORK [initandlisten] waiting for connections
on port 27017
```

Figure 3-3. Starting MongoDB

Installing PHP

A web server is required for running PHP scripts and PHP 5.4, and later versions include a web server packaged in the PHP installation.

1. Download PHP 5.5 (5.5.26) VC11 x64 Thread-Safe version of the PHP zip file [php-5.5.26-Win32-VC11-x64.zip](http://windows.php.net/download/) from <http://windows.php.net/download/>. A later PHP version may also be used. PHP 5.3, 5.4, 5.5 and 5.6 are supported. The VC11 builds require the Visual C++ Redistributable for Visual Studio 2012 x86 or x64 installed as a prerequisite.
2. Extract the [php-5.5.26-Win32-VC11-x64.zip](http://windows.php.net/download/) file to a directory. A [php-5.5.26-Win32-VC11-x64](http://windows.php.net/download/) directory gets created. The directory would be different if a different version is used.
3. Create a document root directory (C:\php used in this chapter) and copy the files and directories from the [php-5.5.26-Win32-VC11-x64](http://windows.php.net/download/) directory to the C:\php directory.
4. Rename the PHP configuration file `php.ini-development` or `php.ini-production` in the root directory of the PHP installation, C:\php, to `php.ini`.
5. Start the packaged web server at port 8000 with the following command from the document root directory C:\php directory.

```
php -S localhost:8000
```

The output from the command indicates that the Development Server has been started and listening on <http://localhost:8000> as shown in Figure 3-4.

```
C:\>php -S localhost:8000
PHP 5.5.26 Development Server started at Fri Jul 31 08:34:31 2015
Listening on http://localhost:8000
Document root is C:\php
Press Ctrl-C to quit.
```

Figure 3-4. Starting PHP Web Server

6. Any PHP script copied to the document root directory (C:\php) may be run on the integrated web server. PHP scripts may be copied to a subdirectory of the document root directory and run by including the directory path starting from the document root in the URL. Copy the following script hellomongo.php to the C:\php directory in the document root.

```
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<?php echo '<p>Hello MongoDB</p>'; ?>
</body>
</html>
```

7. Run the script with the URL <http://localhost:8000/hellomongo.php>. The output is shown in Figure 3-5.

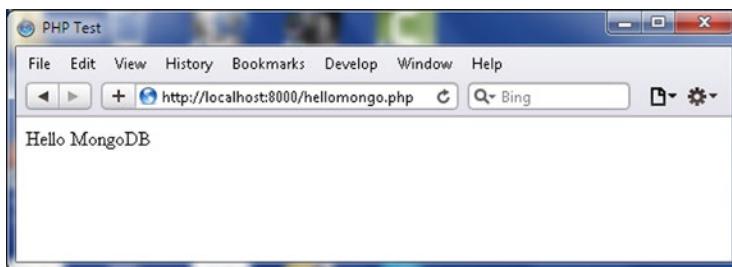


Figure 3-5. Running a PHP Script

Installing PHP Driver for MongoDB

The driver version to download is based on the MongoDB version used. The compatibility matrix for PHP MongoDB Driver and MongoDB version is listed in Table 3-3.

Table 3-3. Compatibility Matrix

PHP Driver	MongoDB Driver
1.6	2.4, 2.6, 3.0
1.5	2.4, 2.6

As we are using MongoDB 3.0.5, the compatible PHP driver version to download is 1.6.

1. Download the PHP MongoDB Database Driver from <http://pecl.php.net/package/mongo>. As we are using PHP 5.5 we need to download the 5.5 Thread Safe (TS) x64 php_mongo-1.6.10-5.5-ts-vc11-x64.zip file from <https://pecl.php.net/package/mongo/1.6.10/windows>.
2. Extract the php_mongo-1.6.10-5.5-ts-vc11-x64.zip file to a directory.
3. Copy the php_mongo.dll file from the extracted directory to the PHP installation document root directory C:\php.
4. Add the following configuration to the php.ini file.

```
extension=php_mongo.dll
```
5. Restart the PHP web server.

Note In subsequent sections we shall run PHP scripts to connect to MongoDB and run CRUD operations in the server. Before each of the subsequent sections (except as noted), run the following command in Mongo shell to delete the catalog collection from the local database; the Mongo shell may be started with the mongo command:

```
use local
db.catalog.drop()
```

We shall be using an empty collection for each of the sections so that document/s from a preceding section are not used, and only the section script is used to demonstrate the PHP MongoDB Driver.

Creating a Connection

Create a PHP script mongoconnection.php in C:\php, the document root. Connect to MongoDB database using the MongoClient constructor as follows.

```
$connection = new MongoClient();
```

Without any connection parameters specified in the constructor, a connection to localhost:27017 is established, localhost being the default MongoDB host and 27017 being the default MongoDB port. Output the connection detail.

```
print 'Connection: <br/>';
var_dump($connection);
```

The following output gets generated.

```
object(MongoClient)#1 (4) { ["connected"]=> bool(true) ["status"]=> NULL
["server":protected]=> NULL ["persistent":protected]=> NULL }
```

The ["connected"]=> bool(true) indicates the client script has connected to MongoDB. The write concern may be output using the getWriteConcern() method as follows.

```
var_dump($connection->getWriteConcern());
```

The following output indicates the value of w as 1 wtimeout as -1.

```
array(2) { ["w"]=> int(1) ["wtimeout"]=> int(-1) }
```

A connection can be obtained using a connection string that must start with `mongodb://`.

```
$connection = new MongoClient( "mongodb://localhost:27017" );
```

Output the read preferences using the getReadPreference() method. The output indicates that the read is directed at the primary member in the replica set, which is also the default.

```
array(1) { ["type"]=> string(7) "primary" }
```

List the databases on the server using the listDBs() method. An array consisting of databases information gets output. For each database the name, sizeOnDisk, empty properties get output. The three databases in the example script are catalog, local and test.

```
array(3) { ["databases"]=> array(3) { [0]=> array(3) { ["name"]=> string(5) "Loc8r"
["sizeOnDisk"]=> float(83886080) ["empty"]=> bool(false) } [1]=> array(3) { ["name"]=> string(5) "local"
["sizeOnDisk"]=> float(83886080) ["empty"]=> bool(false) } [2]=> array(3) { ["name"]=> string(4) "test"
["sizeOnDisk"]=> float(83886080) ["empty"]=> bool(false) } } ["totalSize"]=> float(251658240) ["ok"]=> float(1) }
```

The hostname used in the connection string may also be the IPv4 address as follows. The IPv4 address would be different for different users and may be found using the ipconfig/all command.

```
$connection = new MongoClient("mongodb://192.168.1.72:27017");
```

All the open connections may be listed using the `getConnections()` method. The connections info including host, port, and connection type (STANDALONE for the sample connections) get listed.

```
array(2) { [0]=> array(3) { ["hash"]=> string(24) "localhost:27017;-.;7284" ["server"]=>
array(4) { ["host"]=> string(9) "localhost" ["port"]=> int(27017) ["pid"]=> int(7284)
["version"]=> array(4) { ["major"]=> int(3) ["minor"]=> int(0) ["mini"]=> int(5) ["build"]=>
int(0) } } ["connection"]=> array(12) { ["min_wire_version"]=> int(0) ["max_wire_
version"]=> int(3) ["max_bson_size"]=> int(16777216) ["max_message_size"]=> int(48000000)
["max_write_batch_size"]=> int(1000) ["last_ping"]=> int(1438358245) ["last_ismaster"]=>
int(1438358245) ["ping_ms"]=> int(0) ["connection_type"]=> int(1) ["connection_type_
desc"]=> string(10) "STANDALONE" ["tag_count"]=> int(0) ["tags"]=> array(0) { } } [1]=>
array(3) { ["hash"]=> string(27) "192.168.1.72:27017;-.;7284" ["server"]=> array(4) {
["host"]=> string(12) "192.168.1.72" ["port"]=> int(27017) ["pid"]=> int(7284) ["version"]=>
array(4) { ["major"]=> int(3) ["minor"]=> int(0) ["mini"]=> int(5) ["build"]=> int(0) } }
["connection"]=> array(12) { ["min_wire_version"]=> int(0) ["max_wire_version"]=> int(3)
["max_bson_size"]=> int(16777216) ["max_message_size"]=> int(48000000) ["max_write_batch_
size"]=> int(1000) ["last_ping"]=> int(1438358245) ["last_ismaster"]=> int(1438358245)
["ping_ms"]=> int(0) ["connection_type"]=> int(1) ["connection_type_desc"]=> string(10)
"STANDALONE" ["tag_count"]=> int(0) ["tags"]=> array(0) { } } }
```

To close connection/s invoke the `MongoClient::close ([boolean|string $connection])` method. To close all connections invoke the `close` method with arg true. If an arg is not supplied or is false only the connection on which the method is invoked is closed. A specific connection may be supplied using a connection string. If `getConnections()` method is invoked subsequent to closing all connections an empty array is listed.

```
array(0) { }
```

The PHP script `mongoconnection.php` is listed:

```
<?php
$connection = new MongoClient(); // connects to localhost:27017
print 'Connection: <br/>';
var_dump($connection);
print '<br/>';
print 'Write Concern: <br/>';
var_dump($connection->getWriteConcern());
print '<br/>';
$connection = new MongoClient( "mongodb://localhost:27017" );
print 'Read Preferences: <br/>';
var_dump($connection->getReadPreference());
print '<br/>';
print 'List DBs: <br/>';
var_dump($connection->listDBs());
print '<br/>';
$connection = new MongoClient("mongodb://192.168.1.72:27017");
print 'List Open Connections: <br/>';
var_dump($connection->getConnections());
print '<br/>';
$connection->close(true);
print 'List Open Connections: <br/>';
var_dump($connection->getConnections());
?>
```

Run the PHP script in the browser using the URL `http://localhost:8000/mongoconnection.php`. The output from the `mongoconnection.php` script is shown in Figure 3-6.



```

http://localhost:8000/mongoconnection.php
File Edit View History Bookmarks Develop Window Help
< > + http://localhost:8000/mongoconnection.php Bing
Connection:
object(MongoClient)#1 ( [ "connected" ]=> bool(true) [ "status" ]=> NULL [ "server" ]=> NULL [ "persistent" ]=> NULL )
Write Concern:
array(2) [ "w" ]=> int(1) [ "wtimeout" ]=> int(-1)
Read Preferences:
array(1) [ "type" ]=> string(7) "primary"
List DBs:
array(3) [ "databases" ]=> array(3) ( [ 0 ]=> array(5) ( [ "name" ]=> string(5) "Loc8r" [ "sizeOnDisk" ]=> float(83886080) [ "empty" ]=> bool(false) ) [ 1 ]=> array(5) ( [ "name" ]=> string(5) "local" [ "sizeOnDisk" ]=> float(83886080) [ "empty" ]=> bool(false) ) [ 2 ]=> array(3) ( [ "name" ]=> string(4) "test" [ "sizeOnDisk" ]=> float(83886080) [ "empty" ]=> bool(false) ) ) [ "totalSize" ]=> float(251658240) [ "ok" ]=> float(1)
List Open Connections:
array(2) ( [ 0 ]=> array(3) ( [ "hash" ]=> string(24) "localhost:27017...7284" [ "server" ]=> array(4) ( [ "host" ]=> string(9) "localhost" [ "port" ]=> int(27017) [ "pid" ]=> int(7284) [ "version" ]=> array(4) ( [ "major" ]=> int(3) [ "minor" ]=> int(0) [ "mnin" ]=> int(5) [ "build" ]=> int(0) ) ) [ "connection" ]=> array(12) ( [ "min_wrie_version" ]=> int(0) [ "max_wrie_version" ]=> int(3) [ "max_bson_size" ]=> int(16777216) [ "max_message_size" ]=> int(48000000) [ "max_write_batch_size" ]=> int(1000) [ "last_ping" ]=> int(1438358140) [ "last_imaster" ]=> int(1438358140) [ "ping_ms" ]=> int(0) [ "connection_type" ]=> int(1) [ "connection_type_desc" ]=> string(10) "STANDALONE" [ "tag_count" ]=> int(0) [ "tags" ]=> array(0) ( ) ) [ 1 ]=> array(3) ( [ "hash" ]=> string(27) "192.168.1.72:27017...7284" [ "server" ]=> array(4) ( [ "host" ]=> string(12) "192.168.1.72" [ "port" ]=> int(27017) [ "pid" ]=> int(7284) [ "version" ]=> array(4) ( [ "major" ]=> int(3) [ "minor" ]=> int(0) [ "mnin" ]=> int(5) [ "build" ]=> int(0) ) ) [ "connection" ]=> array(12) ( [ "min_wrie_version" ]=> int(0) [ "max_wrie_version" ]=> int(3) [ "max_bson_size" ]=> int(16777216) [ "max_message_size" ]=> int(48000000) [ "max_write_batch_size" ]=> int(1000) [ "last_ping" ]=> int(1438358140) [ "last_imaster" ]=> int(1438358140) [ "ping_ms" ]=> int(0) [ "connection_type" ]=> int(1) [ "connection_type_desc" ]=> string(10) "STANDALONE" [ "tag_count" ]=> int(0) [ "tags" ]=> array(0) ( ) ) )
List Open Connections:
array(0) (

```

Figure 3-6. Output from the `mongoconnection.php` Script

Getting Database Info

The MongoDB class represents a database. The class instance may be obtained using the `selectDB()` method in `MongoClient` or by direct indirection of a database name. Create a PHP script `db.php` in the `C:\php` directory. Get a MongoDB instance for the test database as follows.

```
$connection = new MongoClient();
$db = $connection->test;
```

Alternatively, the `selectDB()` method may be used. For example, the following MongoDB instance represents the database `local`.

```
$db=$connection->selectDB("local");
```

The `selectDB` method throws the `MongoConnectionException`, which must be handled in a `try-catch` statement. The `getCollectionNames(boolean)` method in MongoDB gets all the connection names in a database. To get the system collections also invoke the method with arg true.

```
$collections=$db->getCollectionNames(true);
```

The collection names may be output as follows.

```
var_dump($collections);
```

The db.php script is listed. MongoDB class does not explicitly appear in the db.php script because the return value is assigned to a variable.

```
<?php
try
{
$connection = new MongoClient();

$db = $connection->test;
print 'Database: ';
var_dump($db);
print '<br/>';
$db=$connection->selectDB("local");
$collections=$db->getCollectionNames(true);
print 'Collections: ';
var_dump($collections);
print '<br/>';
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb, is the "mongo" process running?</p>';
    exit();
}
?>
```

Run the PHP script in a browser with the URL <http://localhost:8000/db.php>. The collection names listed for the local database include system.indexes and catalog as shown in Figure 3-7. The collections listed could be different for different users.

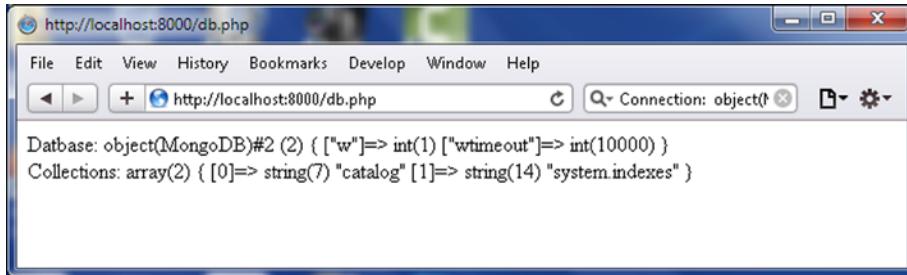


Figure 3-7. Output from the db.php Script

Using Collections

In subsequent subsections we shall discuss getting a collection and dropping a collection using the PHP MongoDB Driver.

Getting a Collection

In this section we shall create a collection in a MongoDB database instance. Create a PHP script `collection.php` in the `C:\php` directory. The `MongoCollection` class represents a collection. The syntax to get a collection from a connection instance is the same as for getting a database. For example first get the database instance `local` and subsequently get the collection `catalog` from the database instance as follows.

```
$connection = new MongoClient();
$db=$connection->local;
$collection=$db->catalog;
```

The collection info and collection name may be output as follows.

```
var_dump($collection);
var_dump($collection->getName());
```

A collection may also be gotten directly as follows.

```
$collection=$connection->local->mongo;
```

The `MongoDB::createCollection()` may also be used to create a `MongoCollection` instance. The `createCollection()` method has the following syntax.

```
$db->createCollection(
    "create" => $name,
    "capped" => $options["capped"],
    "size" => $options["size"],
    "max" => $options["max"],
    "autoIndexId" => $options["autoIndexId"],
);
```

The parameters and options in the `createCollection()` method are as follows in Table 3-4.

Table 3-4. Parameters and Options in `createCollection` Method

Parameter/Option	Description
<code>create</code>	The name of the collection.
<code>capped</code>	Option to indicate if the collection is capped or fixed size.
<code>size</code>	Option to indicate the fixed size of a capped collection in bytes.
<code>max</code>	Option to indicate the maximum number of documents in a capped collection.
<code>autoIndexId</code>	Automatic indexing is true by default for MongoDB 2.2 and later. For pre-2.2 auto indexing is false by default. For capped collections <code>autoIndexId</code> may be set to false.

For example, create a capped collection called `catalog` with fixed size of 1 MB, and maximum number of documents as 10.

```
$coll = $db->createCollection(
    "catalog",
    array(
        'capped' => true,
        'size' => 1*1024,
        'max' => 10
    )
);
```

The `collection.php` script is listed below.

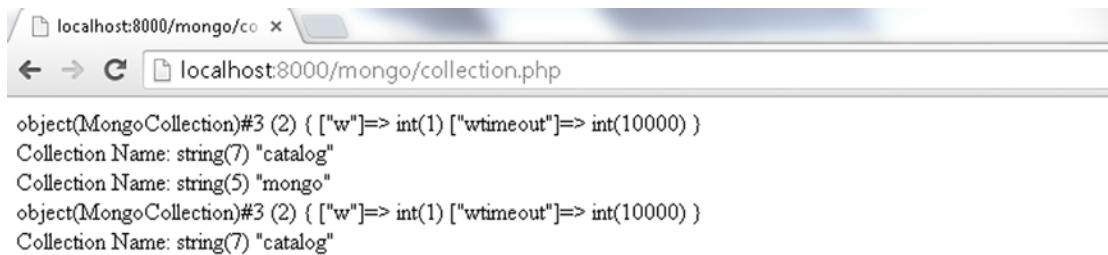
```
<?php
try
{
$connection = new MongoClient();
$db=$connection->local;
$collection=$db->catalog;
var_dump($collection);
print '<br/>';
print 'Collection Name: ';
var_dump($collection->getName());
print '<br/>';
$collection=$connection->local->mongo;
print 'Collection Name: ';
var_dump($collection->getName());

}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb, is the "mongo" process running?</p>';
    exit();
}
$collection->drop();

$coll = $db->createCollection(
    "catalog",
    array(
        'capped' => true,
        'size' => 1*1024,
        'max' => 10
    )
);
print '<br/>';
var_dump($coll);
print '<br/>';
print 'Collection Name: ';
var_dump($coll->getName());

?>
```

Run the `collection.php` script in a browser with the URL `http://localhost:8000/collection.php`. The output is shown in Figure 3-8.



The screenshot shows a browser window with the address bar containing `localhost:8000/mongo/collection.php`. The page content displays the following text:

```
object(MongoCollection)#3 (2) { ["w"]=> int(1) ["wtimeout"]=> int(10000) }
Collection Name: string(7) "catalog"
Collection Name: string(5) "mongo"
object(MongoCollection)#3 (2) { ["w"]=> int(1) ["wtimeout"]=> int(10000) }
Collection Name: string(7) "catalog"
```

Figure 3-8. Output from the `collection.php` Script

Dropping a Collection

The `MongoCollection::drop()` method drops a collection and does not have any parameters.

Create a PHP script `dropCollection.php` in the `C:\php` directory. In a try-catch statement create a `MongoCollection` instance. Invoke the `drop()` method to drop the collection.

```
$collection->drop();
```

The `dropCollection.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$collection->drop();
echo '<p>Collection Dropped</p>';
}catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}
?>
```

Run the PHP script in the browser with URL `http://localhost:8000/dropCollection.php` as shown in Figure 3-9.

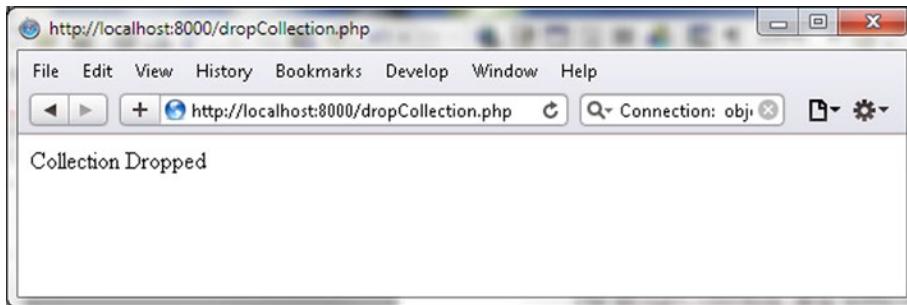


Figure 3-9. Dropping a Collection

The catalog collection gets dropped as indicated by the `show collections` method run before and after running the `dropCollection.php` script in Figure 3-10.

```
> show collections
catalog
system.indexes
> show collections
system.indexes
```

Figure 3-10. Listing Collections Before and After Dropping a Collection

Using Documents

In the following subsections we shall discuss adding, querying, updating, and deleting a document in MongoDB server using the PHP MongoDB Driver.

Adding a Document

The `MongoCollection::insert` method is used to add a single document to MongoDB. The `insert()` method takes parameters.

```
MongoCollection::insert ( array|object $document [, array $options = array() ] )
```

The first parameter is an array or an object and if the parameter does not provide an `_id` key a new `MongoId` instance is created and assigned to it. The `insert` method supports the following options listed in Table 3-5.

Table 3-5. Options in Insert Method

Option	Description
j	To be used if journaling is enabled. Journaling is the process of applying write operations in memory and in the on-disk journal before applying them to data files on disk. If set to true an acknowledged write must be received. The write operation is blocked until it is synced to the journal on disk. The default is false. Overrides a w setting of '0'.
fsync	If journaling is enabled fsync is similar to j. If journaling is not enabled the write operation is blocked until it is synced with data files on disk and an acknowledged insert must be received overriding a w setting of '0'.
socketTimeoutMS	Specifies the time limit for socket communication and a MongoCursorTimeoutException exception is thrown if the server does not respond within this time period. The default value for MongoClient is 30,000 ms.
w	Write concern with a default value of 1.
wTimeoutMS	For w>1 specifies the time limit in ms for acknowledgement. If write concern is not met within the time limit a MongoCursorException exception is thrown. The default value for MongoClient is 10,000.

1. Create a PHP script addDocument.php in the C:\php directory. In a try-catch statement create a MongoClient instance, which represents a connection, and create a MongoCollection instance in local database for catalog collection.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Create an array for a document to add.

```
$doc = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Engineering as a Service', "author" => 'David A. Kelly')
);
```

3. Add the document to the MongoDB collection using the insert() method.

```
$status=$collection->insert($doc);
var_dump($status);
```

4. Add catch blocks for `MongoConnectionException` and `MongoCursorException`.
5. Similarly add another document to the collection. The `addDocument.php` script is listed below.

```
<?php
try
{
$connection = new MongoClient();

$collection=$connection->local->catalog;
$doc = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Engineering as a Service', "author" => 'David A. Kelly')
);
$status=$collection->insert($doc);
var_dump($status);
print '<br/>';
$doc = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Quintessential and Collaborative', "author" => 'Tom Haunert')
);
$status=$collection->insert($doc);
var_dump($status);
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>
```

6. Run the PHP script in the browser with the URL `http://localhost:8000/addDocument.php`. The output is shown in Figure 3-11. As indicated by the `status` message array returned by the `insert()` method the `err` key is `NULL`, which implies no error and the `errmsg` is also `NULL`. And the `ok` key is `1`, which implies all database operations completed successfully. The `n` key indicates the number of documents affected, which is `0` for an insert. The `n` for an update, upsert, or remove would be a positive number if document/s have been updated, upserted, or removed.

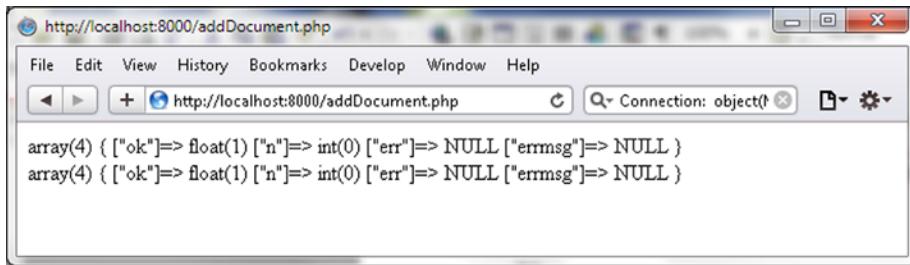


Figure 3-11. Output from the addDocument.php Script

As we did not provide a `_id` key for the documents added, a unique `MongoId` is created automatically for the documents added. The `_id` key or `MongoId` must be unique. Next, we shall demonstrate that the `_id` or `MongoId` must be unique.

Using a try-catch statement create and add a document using the `insert()` method. Invoke the `insert` method on the same document twice.

```
$status=$collection->insert($doc);
$status=$collection->insert($doc);
```

The `addDocumentException.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();

$collection=$connection->local->catalog;
$doc = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" =>
'Engineering as a Service', "author" => 'David A. Kelly')
);
$status=$collection->insert($doc);
var_dump($status);
print '<br/>';

$status=$collection->insert($doc);
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}

?>
```

When the `addDocumentException.php` script is run in a browser (URL `http://localhost:8000/addDocumentException.php`) the `MongoCursorException` exception is thrown as the same document is being tried to be added twice. The exception is caught and an error message is output as shown in Figure 3-12.

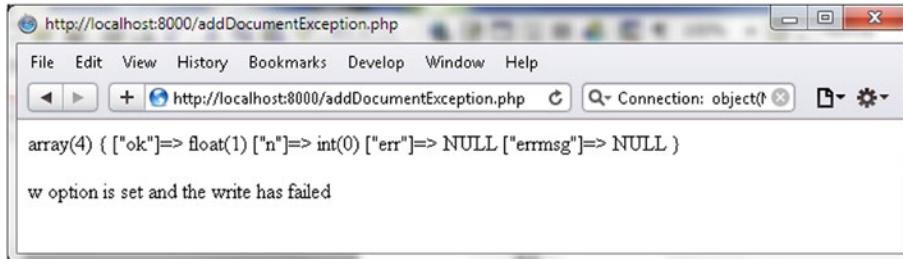


Figure 3-12. Output from the `addDocumentException.php` Script

Adding Multiple Documents

In the preceding section we added two documents using two separate invocations of the `insert()` method. The `insert()` method invocations may also be made using a `while`, `do-while`, or `for` loop.

1. Create a PHP script `addMultipleDocuments.php` in the document root directory `C:\php`. In a try-catch statement create an instance of `MongoCollection` for a collection as before.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. In a `for` loop add documents to the collection using the variable `$i` from the `for` loop initializer in the `catalogId` field for the documents.

```
for ($i = 1; $i <= 5; $i++)
{
    $status=$collection->insert(array("catalogId" =>'catalog' . $i, "journal" =>
'Oracle Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November
December 2013'));
var_dump($status);
print '<br/>';
}
```

The `addMultipleDocuments.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
for ($i = 1; $i <= 5; $i++)
{
```

```

$status=$collection->insert(array("catalogId" =>'catalog' . $i, "journal" =>
'Oracle Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November
December 2013'));
var_dump($status);
print '<br/>';
}
}catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>
```

- Run the `addMultipleDocuments.php` script in a browser with URL `http://localhost:8000/addMultipleDocuments.php` to add multiple documents to the MongoDB collection. A status message gets output for each document added as shown in Figure 3-13.

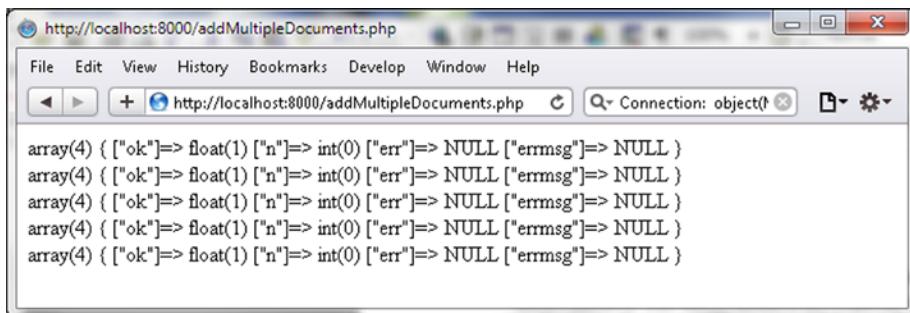


Figure 3-13. Output from the `addMultipleDocuments.php` Script

Adding a Batch of Documents

The `MongoCollection::batchInsert()` method may be used to add a batch of documents to a collection. The method syntax is the same as the `insert` method.

```
MongoCollection::batchInsert ( array $document [, array $options = array() ] )
```

The difference from the `insert()` method is that the first parameter `$document` is of type array of arrays or objects instead of array or object in the `insert` method. The `batchInsert()` method returns an associative array instead of the array or boolean for the `insert` method. The method supports one additional option, the `continueOnError` option, which defaults to false. If set to true the bulk insert fails if `insert` for one of the documents fails. Each of the documents in the array of documents to be added in a batch must have a unique `_id`.

1. Create a PHP script addDocumentBatch.php in the C:\php directory. Create a MongoCollection instance as before.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Create an array to which the document arrays are to be added.

```
$batch=array();
```

3. Create a document array and add the array to the \$batch array.

```
$doc1 = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog1', "journal" => 'Oracle
Magazine', "publisher" => 'Oracle Publishing', "edition" => 'November December
2013', "title" => 'Engineering as a Service', "author" => 'David A. Kelly')
);

$batch[]=$doc1;
```

4. Similarly add another document to the batch array. The \$doc2 is declared in the addDocumentBatch.php script listing.

```
$batch[]=$doc2;
```

5. Invoke the batchInsert method with the \$batch array as arg.

```
$status=$collection->batchInsert($batch);
var_dump($status);
```

6. As the _id fields for the documents to be added are not provided the MongoDB instances are added automatically. Subsequently the _id field values added may be output in a foreach loop.

```
foreach ($batch as $doc) {
print 'Document _id: ';
echo $doc['_id']."\n";
print '<br/>';
}
```

The addDocumentBatch.php script is listed.

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$batch=array();
```

```

$doc1 = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013',"title" =>
'Engineering as a Service','author" => 'David A. Kelly')
);

$batch[ ]=$doc1;

$doc2 = array(
    "name" => "MongoDB",
    "type" => "database",
    "count" => 1,
    "info" => (object)array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013',"title" =>
'Quintessential and Collaborative','author" => 'Tom Haunert')
);

$batch[ ]=$doc2;
$status=$collection->batchInsert($batch);
var_dump($status);
print '<br/>';
foreach ($batch as $doc) {
print 'Document _id: ';
echo $doc['_id']."\n";
print '<br/>';
}

}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}

}
?>
```

- Run the PHP script `addDocumentBatch.php` in the browser with URL `http://localhost:8000/addDocumentBatch.php`. As indicated by the output in Figure 3-14, two documents with unique ids get added to MongoDB collection catalog.

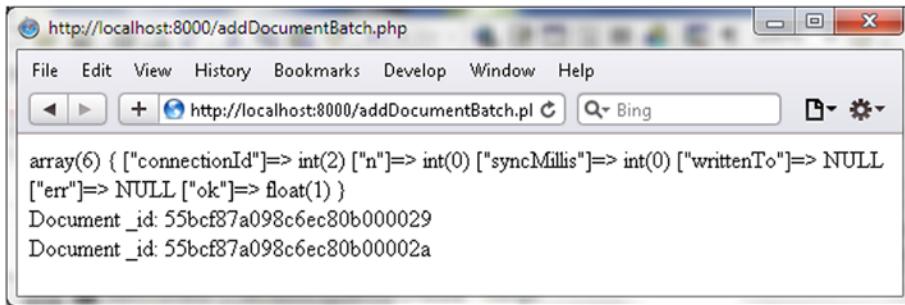


Figure 3-14. Output from the *addDocumentBatch.php* Script

- Subsequently invoke the `find()` method for the `catalog` collection in local database in the Mongo shell to output the documents added.

```
>use local
>db.catalog.find()
```

The two documents added get listed as shown in Figure 3-15.



Figure 3-15. Listing Documents Added in a Batch in Mongo Shell

Do not delete the `catalog` collection in the local database as the documents added in a batch shall be used to demonstrate finding a document in the next section.

Finding a Single Document

The `MongoCollection::findOne()` method is used to find a single document from a collection. The `findOne()` method returns an array consisting of the fields of the document. The method takes as parameters a `$query`, and the `$fields` to include in the document returned, and options. The `_id` field is returned even if not specified in `$fields`.

```
MongoCollection::findOne ([ array $query = array() [, array $fields = array() [, array $options = array() ]]] )
```

All parameters are optional and if none are specified the first document from the collection is returned. Only one option is supported, `maxTimeMS`, which is the cumulative time limit in ms for processing the method not including the idle time. If the method does not complete in specified time a `MongoExecutionTimeoutException` is thrown. The `MongoConnectionException` is thrown if a connection with the MongoDB server is not established.

1. Create a PHP script `findDocument.php` in the `C:\php` directory. In a try-catch statement, create a `MongoClient` instance and using the connection create a `MongoCollection` instance for the `catalog` collection in the `local` database.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Invoke the `findOne()` method on the `MongoCollection` instance to find a single document. The first document found is returned and could be different for different users.

```
$document = $collection->findOne();
var_dump($document);
```

3. A specific document may be found by specifying the `_id` field in the array supplied to the `findOne()` method. The `_id` field value is constructed using the `MongoId` class constructor. The `_id` field value would be different for different users.

```
$document = $collection->findOne(array('_id' => new MongoId
("55bcf87a098c6ec80b00002a")));
var_dump($document);
```

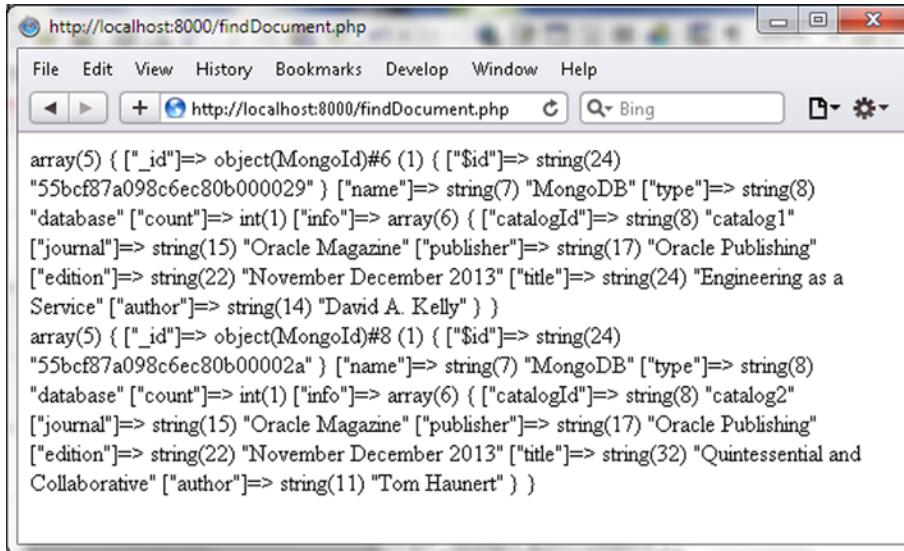
The PHP script `findDocument.php` is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;

$document = $collection->findOne();
var_dump($document);
print '<br/>';
$document = $collection->findOne(array('_id' => new MongoId("55bcf87a098c6ec80b00002a")));
var_dump($document);
print '<br/>';

}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>
```

4. Run the PHP script in a browser with the URL `http://localhost:8000/findDocument.php`. Two documents get displayed in the browser as shown in Figure 3-16.



The screenshot shows a Microsoft Internet Explorer browser window with the URL `http://localhost:8000/findDocument.php` in the address bar. The page content displays two JSON objects representing documents from a MongoDB collection. The first document has an _id of `55bcf87a098c6ec80b000029`, a name of "MongoDB", a type of "string(8)", and a catalog named "catalog1". The second document has an _id of `55bcf87a098c6ec80b00002a`, a name of "MongoDB", a type of "string(8)", and a catalog named "catalog2". Both documents have a journal of "Oracle Magazine", a publisher of "Oracle Publishing", an edition of "November December 2013", and a title of "Engineering as a Service" or "Quintessential and Collaborative" respectively, with an author of "David A. Kelly" or "Tom Haunert".

```

array(5) { ["_id"]=> object(MongoId)#6 (1) { ["$id"]=> string(24) "55bcf87a098c6ec80b000029" } ["name"]=> string(7) "MongoDB" ["type"]=> string(8) "database" ["count"]=> int(1) ["info"]=> array(6) { ["catalogId"]=> string(8) "catalog1" ["journal"]=> string(15) "Oracle Magazine" ["publisher"]=> string(17) "Oracle Publishing" ["edition"]=> string(22) "November December 2013" ["title"]=> string(24) "Engineering as a Service" ["author"]=> string(14) "David A. Kelly" } }
array(5) { ["_id"]=> object(MongoId)#8 (1) { ["$id"]=> string(24) "55bcf87a098c6ec80b00002a" } ["name"]=> string(7) "MongoDB" ["type"]=> string(8) "database" ["count"]=> int(1) ["info"]=> array(6) { ["catalogId"]=> string(8) "catalog2" ["journal"]=> string(15) "Oracle Magazine" ["publisher"]=> string(17) "Oracle Publishing" ["edition"]=> string(22) "November December 2013" ["title"]=> string(32) "Quintessential and Collaborative" ["author"]=> string(11) "Tom Haunert" } }

```

Figure 3-16. Finding a Single Document with `findOne()`

Again, do not delete the `catalog` collection in the local database as the documents added in batches shall be used to demonstrate finding documents in the next section.

Finding All Documents

The `MongoCollection::find()` method is used to find all documents that match a specified query. The method syntax takes two parameters, `$query` and `$fields`, both of type array and both optional. The `_id` field is always returned. If a query is not specified all documents are returned. The `find()` method returns a cursor, represented by a `MongoCursor` instance, over the result set of the database query.

```
MongoCursor MongoCollection::find ([ array $query = array() [, array $fields = array() ] ] )
```

1. Create a PHP script `findAllDocuments.php` in the `C:\php` directory. In a `try-catch` statement create a `MongoClient` instance, which represents a connection with the MongoDB server. Create a `MongoCollection` instance for the `catalog` collection in the local database.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Invoke the `find()` method on the `MongoCollection` instance and use the `iterator_to_array` method to convert the cursor returned to an array.

```
$cursor = $collection->find();
var_dump(iterator_to_array($cursor));
```

3. The foreach loop may also be used to iterate over the result set of the database query. For example the `_id` and `catalogId` field values are output as follows.

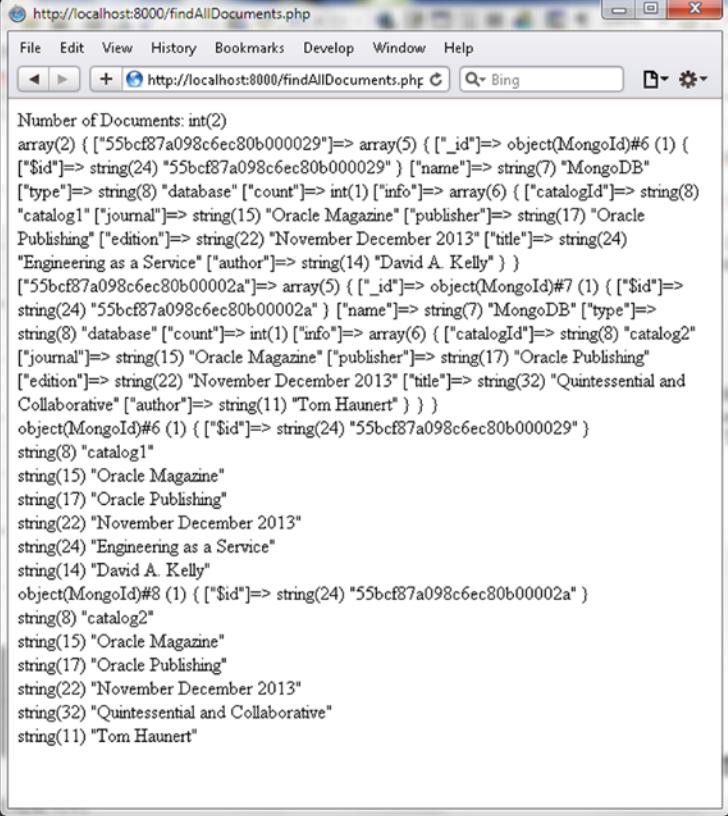
```
foreach ($cursor as $doc) {
    var_dump($doc["_id"]);
    print '<br/>';
    var_dump($doc["info"]["catalogId"]);
}
```

The `findAllDocuments.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
print 'Number of Documents: ';
var_dump($collection->count());
print '<br/>';

$cursor = $collection->find();
var_dump(iterator_to_array($cursor));
print '<br/>';
foreach ($cursor as $doc) {
    var_dump($doc["_id"]);
    print '<br/>';
    var_dump($doc["info"]["catalogId"]);
    print '<br/>';
    var_dump($doc["info"]["journal"]);
    print '<br/>';
    var_dump($doc["info"]["publisher"]);
    print '<br/>';
    var_dump($doc["info"]["edition"]);
    print '<br/>';
    var_dump($doc["info"]["title"]);
    print '<br/>';
    var_dump($doc["info"]["author"]);
    print '<br/>';
}
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>
```

4. Run the PHP script in a browser with the URL `http://localhost:8000/findAllDocuments.php`. All the documents in the `catalog` collection in the local database get displayed. The field values for each of the documents also get displayed as shown in Figure 3-17.



```

Number of Documents: int(2)
array(2) { ["55bcf87a098c6ec80b000029"]=> array(5) { ["_id"]=> object(MongoId)#6 (1) {
    ["$id"]=> string(24) "55bcf87a098c6ec80b000029" } ["name"]=> string(7) "MongoDB"
    ["type"]=> string(8) "database" ["count"]=> int(1) ["info"]=> array(6) { ["catalogId"]=> string(8)
    "catalog1" ["journal"]=> string(15) "Oracle Magazine" ["publisher"]=> string(17) "Oracle
    Publishing" ["edition"]=> string(22) "November December 2013" ["title"]=> string(24)
    "Engineering as a Service" ["author"]=> string(14) "David A. Kelly" )
    ["55bcf87a098c6ec80b00002a"]=> array(5) { ["_id"]=> object(MongoId)#7 (1) { ["$id"]=>
    string(24) "55bcf87a098c6ec80b00002a" } ["name"]=> string(7) "MongoDB" ["type"]=>
    string(8) "database" ["count"]=> int(1) ["info"]=> array(6) { ["catalogId"]=> string(8) "catalog2"
    ["journal"]=> string(15) "Oracle Magazine" ["publisher"]=> string(17) "Oracle Publishing"
    ["edition"]=> string(22) "November December 2013" ["title"]=> string(32) "Quintessential and
    Collaborative" ["author"]=> string(11) "Tom Haunert" ) )
    object(MongoId)#6 (1) { ["$id"]=> string(24) "55bcf87a098c6ec80b000029" }
    string(8) "catalog1"
    string(15) "Oracle Magazine"
    string(17) "Oracle Publishing"
    string(22) "November December 2013"
    string(24) "Engineering as a Service"
    string(14) "David A. Kelly"
    object(MongoId)#8 (1) { ["$id"]=> string(24) "55bcf87a098c6ec80b00002a" }
    string(8) "catalog2"
    string(15) "Oracle Magazine"
    string(17) "Oracle Publishing"
    string(22) "November December 2013"
    string(32) "Quintessential and Collaborative"
    string(11) "Tom Haunert"
}

```

Figure 3-17. Finding All Documents with `find()`

Finding a Subset of Fields and Documents

The `find()` method takes two parameters of type array, `$query` and `$fields`, both of which are optional. To select a subset of fields from a subset of documents parameter values for both may be specified.

1. First, add a document set using the following script, `addDocumentSet.php` in the `C:\php` directory.

```

<?php
try
{
    $connection = new MongoClient();
    $collection=$connection->local->catalog;
    $doc = array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine',
    "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title"
    => 'Engineering as
    Service', "author" => 'David A. Kelly');
    $status=$collection->insert($doc);
    var_dump($status);
    print '<br/>';
}

```

```

$doc = array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013',"title"
=> 'Quintessential and Collaborative',"author" => 'Tom Haunert');
$status=$collection->insert($doc);
var_dump($status);
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>

```

2. Copy the script to the C:\php directory and run with URL
<http://localhost:8000/addDocumentSet.php>.
3. Create another PHP script, `findDocumentSet.php`, in the C:\php directory to find a subset of fields and documents.
4. Create a `MongoCollection` instance for `catalog` collection in the local database as before.

```

$connection = new MongoClient();
$collection=$connection->local->catalog;

```

5. Specify the array of key=>value pairs for which documents are to be found. As an example select all documents with `catalogId` as `catalog1`.

```
$query = array('catalogId'=>'catalog1');
```

6. Specify an array of fields to select from the document/s found. As an example select the `title` and `author` fields.

```
$fields = array('title' => true, 'author' => true);
```

7. Invoke the `find()` method using the `$query` and `$fields` args to get a cursor over the result set.

```
$cursor = $collection->find($query, $fields);
```

8. Using a `while` loop iterate over the result to output the document fields returned. The `hasNext()` method in `MongoCursor` moves the cursor to the next document and the `getNext()` method gets the next document.

```

while ($cursor->hasNext())
{
    var_dump($cursor->getNext());
}
?>

```

The `findDocumentSet.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$query = array('catalogId'=>'catalog1');
$fields = array('title' => true, 'author' => true);
$cursor = $collection->find($query, $fields);
while ($cursor->hasNext())
{
    var_dump($cursor->getNext());
}
}catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo $e;
    exit();
}
?>
```

- Run the `findDocumentSet.php` script in a browser with URL

`http://localhost:8000/findDocumentSet.php` to output the selected fields from the selected documents as shown in Figure 3-18.

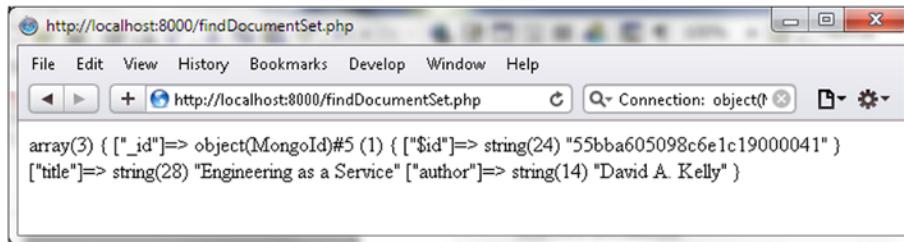


Figure 3-18. Finding Subset of Fields for Subset of Documents

- To select all documents specify the query as follows.

```
$query = array();
```

When the `findDocumentSet.php` script is run in the browser to select all documents, the selected fields from all documents get selected as shown in Figure 3-19.

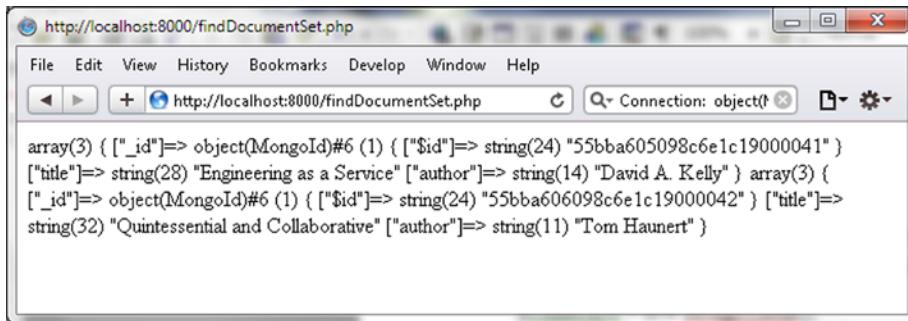


Figure 3-19. Finding Subset of Fields for All Documents

Updating a Document

The `MongoCollection::update()` method is used to update one or more documents based on a specified criteria. The method returns a status array if `w` option is set or returns a boolean. The method syntax takes the `$criteria`, `$new_object` and `$options` parameters all of type array.

```
MongoCollection::update ( array $criteria , array $new_object [, array $options = array() ] )
```

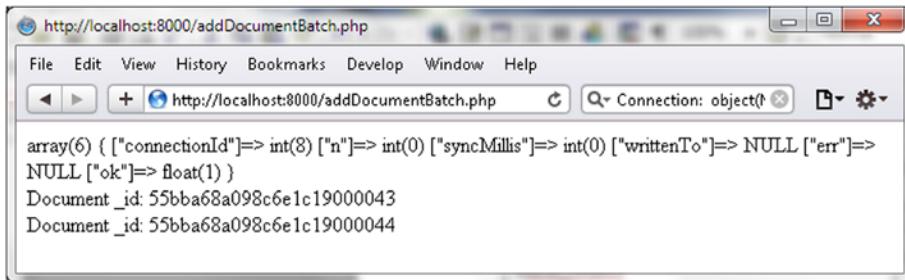
The method parameters are discussed in Table 3-6.

Table 3-6. Parameters for the Update Method

Parameter	Description
<code>\$criteria</code>	Query criteria for the documents to update.
<code>\$new_object</code>	The replacement document if the new object contains key=>value pairs. Or if the new object contains update operators the specific fields to update.
<code>\$options</code>	The main options are <code>upsert</code> , <code>w</code> , and <code>multiple</code> . The <code>upsert</code> option if set to true adds a new document if no document matches criteria. The default value of <code>upsert</code> is false. The default value of <code>w</code> is 1. The <code>multiple</code> option if set to true updates multiple documents. Default value of <code>multiple</code> is false.

Next, we shall update some documents.

1. First, we need to add the documents to update. Run the `addDocumentBatch.php` script with URL `http://localhost:8000/addDocumentBatch.php` as shown in Figure 3-20 to add two documents to the `catalog` collection in the `local` database. The document `_id` is output. We shall use these `_id` values to update the documents. The `_id` values would be different for different users.

**Figure 3-20.** Adding a Batch of Documents

2. Create a PHP script `updateDocument.php` in the `C:\php` directory. In a try-catch statement create a connection with MongoDB using a `MongoClient` instance. Create a `MongoCollection` instance for the `catalog` collection in the local database.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

3. Specify the `$criteria` for the document to update by setting the `_id` field in the array to a `MongoId` instance constructed from `53f3d425098c6e2410000065`, which is the `_id` for one of the documents added to the `catalog` collection with `addDocumentBatch.php`. The `_id` value would be different for different users.

```
$criteria = array("_id" => new MongoId("55bba68a098c6e1c19000043 "));
```

4. Specify a `$new_object` using key=>value pairs for the replacement document. Add a new field “`updated`” set to “`true`.”

```
$new_object = array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => '11-12-2013', "title" =>
'Engineering As a Service', "author" => 'Kelly, David A.', "updated"=>true);
```

5. Invoke the `update()` method using the `$criteria` and `$new_object` and using an options array with `upsert` set to `false`.

```
$status=$collection->update($criteria,$new_object, array("upsert" => false));
var_dump($status);
```

6. Similarly, update another document.

```
$criteria = array("_id" => new MongoId("55bba68a098c6e1c19000044"));
$new_object = array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => '11-12-2013', "title" =>
'Quintessential and Collaborative', "author" => 'Haunert, Tom', "updated"=>true);
$status=$collection->update($criteria,$new_object, array("upsert" => false));
var_dump($status);
```

We set the `upsert` option to `false` in the preceding document updates. Next we shall invoke the `update` method using `upsert` option set to `true`.

To demonstrate that upsert adds new document if a document for the \$criteria is not found specify a \$criteria using a _id that does not already exist in the database. The same _id value may be used by different users as in the updateDocument.php script listed.

```
$criteria = array("_id" => new MongoDB\BSON\ObjectId("53f3d425098c6e2410000064"));
```

7. Specify a \$new_object replacement document and invoke the update() method with upsert set to true.

```
$new_object = array("catalogId" => 'catalog3', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => '11-12-2013');
$status=$collection->update($criteria,$new_object, array("upsert" => true));
```

The updateDocument.php script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$criteria = array("_id" => new MongoDB\BSON\ObjectId("55bba68a098c6e1c19000043"));

$new_object = array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine', "publisher"
=> 'Oracle Publishing', "edition" => '11-12-2013','title' => 'Engineering As a
Service','author' => 'Kelly, David A.', "updated"=>true);
$status=$collection->update($criteria,$new_object, array("upsert" => false));
var_dump($status);
print '<br/>';
$criteria = array("_id" => new MongoDB\BSON\ObjectId("55bba68a098c6e1c19000044"));
$new_object = array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine', "publisher"
=> 'Oracle Publishing', "edition" => '11-12-2013','title' => 'Quintessential and
Collaborative','author' => 'Haunert, Tom', "updated"=>true);
$status=$collection->update($criteria,$new_object, array("upsert" => false));
var_dump($status);
print '<br/>';
$criteria = array("_id" => new MongoDB\BSON\ObjectId("53f3d425098c6e2410000064"));
$new_object = array("catalogId" => 'catalog3', "journal" => 'Oracle Magazine', "publisher"
=> 'Oracle Publishing', "edition" => '11-12-2013');
$status=$collection->update($criteria,$new_object, array("upsert" => true));
var_dump($status);
}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>
```

8. Run the `updateDocument.php` script in a browser using URL `http://localhost:8000/updateDocument.php`. Two documents get updated and one document gets upserted. The `updatedExisting` key is true in two of the status arrays and false in one status array as shown in Figure 3-21.

```

array(6) { ["ok"]=> float(1) ["nModified"]=> int(1) ["n"]=> int(1) ["err"]=> NULL ["errmsg"]=> NULL ["updatedExisting"]=> bool(true)
array(6) { ["ok"]=> float(1) ["nModified"]=> int(1) ["n"]=> int(1) ["err"]=> NULL ["errmsg"]=> NULL ["updatedExisting"]=> bool(true)
array(7) { ["ok"]=> float(1) ["nModified"]=> int(0) ["n"]=> int(1) ["err"]=> NULL ["errmsg"]=> NULL ["upserted"]=> object(MongoId)#5 (1) { ["$id"]=> string(24)
"53f3d425098c6e2410000064" } ["updatedExisting"]=> bool(false)

```

Figure 3-21. Updating a Document

9. Run the following JavaScript method in Mongo shell

```

>use local
>db.catalog.find()

```

The updated/upserted documents get listed as shown in Figure 3-22.

```

> use local
switched to db local
> db.catalog.find()
[{"_id": ObjectId("55bba68a098c6e1c19000043"), "name": "MongoDB", "type": "database", "count": 1, "info": {"catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, "catalog2": {"catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}, "catalog3": {"catalogId": "catalog3", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "11-12-2013"}]
>

```

Figure 3-22. Listing Updated Documents

Updating Multiple Documents

In the preceding section we mentioned the `multiple` option in the `update()` method to update multiple documents. In this section we shall use the `multiple` option.

1. Create a PHP script `updateMultiDocuments.php` in the `C:\php` directory. In a try-catch statement create a `MongoCollection` instance as before.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Add two documents using the `insert()` method. Do not include the `journal` field in the documents added as we shall be adding the field using the `update()` method.

```
$collection->insert(array("catalogId" => 'catalog1', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Engineering as a Service', "author" => 'David A. Kelly'));
$collection->insert(array("catalogId" => 'catalog2', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Quintessential and Collaborative', "author" => 'Tom Haunert'));
```

3. We shall add the `journal` field to the documents added using the update operator `$set` in the `update` method with the `multiple` option set to true.

```
$newdata = array('$set' => array("journal" => "Oracle Magazine"));
```

4. Invoke the `update()` method using the `$criteria` as documents with edition field as November December 2013, the `$newdata` for fields to update, and options array with `multiple` set to true.

```
$status=$collection->update(array("edition" => "November December 2013"),
$newdata, array("multiple" => true));
```

The `updateMultiDocuments.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;

$collection->insert(array("catalogId" => 'catalog1', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Engineering as a Service', "author" => 'David A. Kelly'));

$collection->insert(array("catalogId" => 'catalog2', "publisher" => 'Oracle Publishing', "edition" => 'November December 2013', "title" => 'Quintessential and Collaborative', "author" => 'Tom Haunert'));

$newdata = array('$set' => array("journal" => "Oracle Magazine"));
$status=$collection->update(array("edition" => "November December 2013"),
$newdata, array("multiple" => true));
```

```

var_dump($status);

}

catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
} catch(MongoCursorException $e) {
    echo $e;
    exit();
}
?>

```

- Run the updateMultiDocuments.php script in a browser using the URL <http://localhost:8000/updateMultiDocuments.php>. In the output updatedExisting key value is true with nModified key value as 2, which indicates that two existing documents got updated as shown in Figure 3-23.

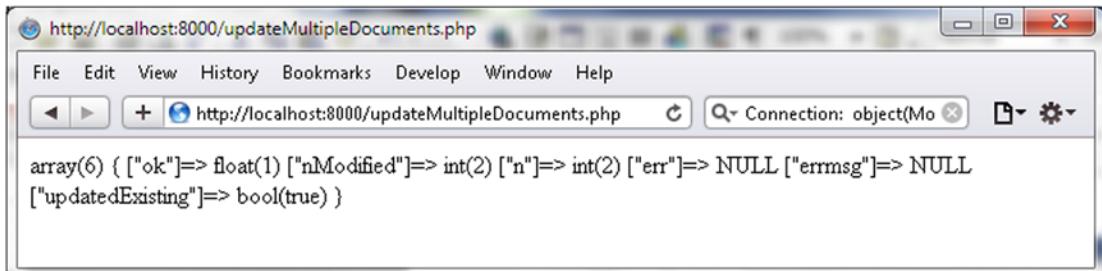


Figure 3-23. Updating Multiple Documents

- Run the db.catalog.find() method in Mongo shell to list the updated documents. As shown in the output the documents include the journal field as shown in Figure 3-24.

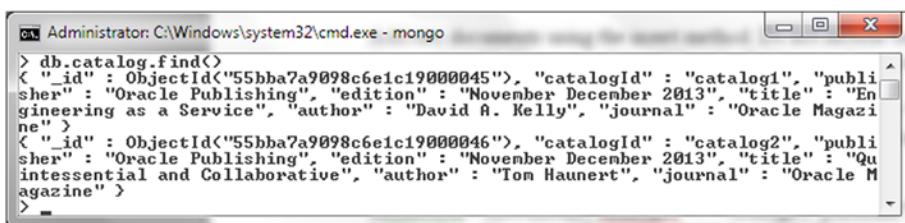


Figure 3-24. Listing Updated Documents

Saving a Document

By default the `insert()` method does not add a modified document if the document already exists in the database. The `MongoCollection` class provides another method to insert a modified document. The `MongoCollection::save()` method saves a document to a collection. `Save` is different from `insert` in that

the document to be saved may already exist in the database. In this section we shall add two documents using the `insert()` method and subsequently invoke the `save()` method to save the same documents with modified field values for some of the fields. The syntax of the `save` method is as follows.

```
MongoCollection::save ( array|object $document [, array $options = array() ] )
```

1. Create a PHP script `saveDocument.php` in the `C:\php` directory. In the try-catch statement create a `MongoCollection` instance.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

2. Add two documents using the `insert()` method.

```
$doc1 = array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013',"title"
=> 'Engineering as a
Service',"author" => 'David A. Kelly');
$status=$collection->insert($doc1);
$doc2 =array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine',
"publisher" => 'Oracle Publishing', "edition" => 'November December 2013',"title"
=> 'Quintessential and Collaborative',"author" => 'Tom Haunert');
$status=$collection->insert($doc2);
```

3. Subsequently modify some of the field values in the two documents and invoke the `save()` method on the modified documents.

```
$doc1['edition']= '11-12-2013';
$doc1['author'] = 'Kelly, David A.';
$doc1['updated']=true;
$status=$collection->save($doc1);
$doc2['edition']= '11-12-2013';
$doc2['author'] = 'Haunert, Tom';
$doc2['updated']=true;
$status=$collection->save($doc2);
```

The `save()` method saves the documents with the modified field values. If we had used the `insert` method to save the modified documents we would have received an error. The `saveDocument.php` script is listed:

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$doc1 = array("catalogId" => 'catalog1', "journal" => 'Oracle Magazine', "publisher" =>
'Oracle
Publishing', "edition" => 'November December 2013',"title" => 'Engineering as a
Service',"author" => 'David A. Kelly');
$status=$collection->insert($doc1);
var_dump($status);
```

```

print '<br/>';
$doc2 =array("catalogId" => 'catalog2', "journal" => 'Oracle Magazine', "publisher" =>
'Oracle Publishing', "edition" => 'November December 2013',"title" => 'Quintessential and
Collaborative','author' => 'Tom Haunert');
$status=$collection->insert($doc2);
var_dump($status);
print '<br/>';
$doc1['edition']= '11-12-2013';
$doc1['author'] = 'Kelly, David A.';
$doc1['updated']=true;

$status=$collection->save($doc1);
var_dump($status);
print '<br/>';

$doc2['edition']= '11-12-2013';
$doc2['author'] = 'Haunert, Tom';
$doc2['updated']=true;
$status=$collection->save($doc2);
var_dump($status);
print '<br/>';

}catch ( MongoConnectionException $e )
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}catch(MongoCursorException $e) {
    echo '<p>w option is set and the write has failed</p>';
    exit();
}
?>

```

- Run the PHP script in the browser with the URL <http://localhost:8000/saveDocument.php>. Two documents get added. Subsequently the two documents get updated as indicated by the updatedExisting key value true in Figure 3-25.

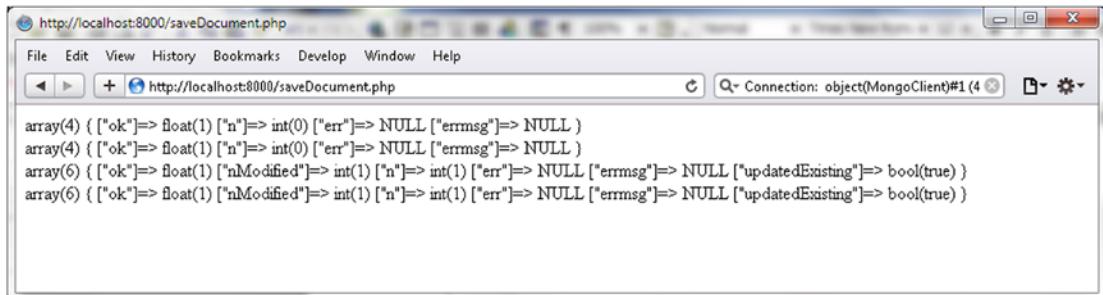


Figure 3-25. Saving Documents

- Run the JavaScript method `db.catalog.find()` in Mongo shell to list the updated documents as shown in Figure 3-26.

```
> db.catalog.find()
{ "_id" : ObjectId("55bba861098c6e1c19000047"), "catalogId" : "catalog1", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013", "title" : "Engineering as a Service", "author" : "Kelly, David A.", "updated" : true }
{ "_id" : ObjectId("55bba861098c6e1c19000048"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013", "title" : "Quintessential and Collaborative", "author" : "Haunert, Tom", "updated" : true }
>
```

Figure 3-26. Listing Saved Documents

Removing a Document

The MongoCollection::remove() method to remove a document has the following syntax.

```
MongoCollection::remove ([ array $criteria = array() [, array $options = array() ]] )
```

The method parameter \$criteria is the query criteria for the document/s to remove. Most of the options such as w, j, fsync, are the same as the other methods. The remove() method provides the justOne option to remove just one document. In this section we shall remove a document from a collection.

1. First, add some documents using the addDocumentBatch.php script as shown in Figure 3-27.

```
array(6) { ["connectionId"]=> int(8) ["n"]=> int(0) ["syncMillis"]=> int(0) ["writtenTo"]=> NULL ["err"]=> NULL ["ok"]=> float(1) }
Document _id: 55bba8d9098c6e1c19000049
Document _id: 55bba8d9098c6e1c1900004a
```

Figure 3-27. Adding Documents with addDocumentBatch.php

The db.catalog.find() method in mongo shell should list the two documents added as shown in Figure 3-28.

```
> db.catalog.find()
{ "_id" : ObjectId("55bba8d9098c6e1c19000049"), "name" : "MongoDB", "type" : "database", "count" : 1, "info" : { "catalogId" : "catalog1", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" } }
{ "_id" : ObjectId("55bba8d9098c6e1c1900004a"), "name" : "MongoDB", "type" : "database", "count" : 1, "info" : { "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" } }
```

Figure 3-28. Listing Documents added with addDocumentBatch.php

2. Create a PHP script `removeDocument.php` in the `C:\php` directory. Create a `MongoCollection` instance as before.

```
$connection = new MongoClient();
$collection=$connection->local->catalog;
```

3. Invoke the `remove` method with the `_id` for the document to remove as method arg. The `_id` field must be supplied as a `MongoId` instance and would be different for different users.

```
$id = '55bba8d9098c6e1c19000049';
$status=$collection->remove(array('_id' => new MongoId($id)));
```

The `removeDocument.php` script is listed:

```
<?php
try
{
$id = '55bba8d9098c6e1c19000049';
$connection = new MongoClient();
$collection=$connection->local->catalog;
$status=$collection->remove(array('_id' => new MongoId($id)));
var_dump($status);
}catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}
?>
```

4. Run the `removeDocument.php` script in a browser with URL `http://localhost:8000/removeDocument.php` to remove a document as shown in Figure 3-29.

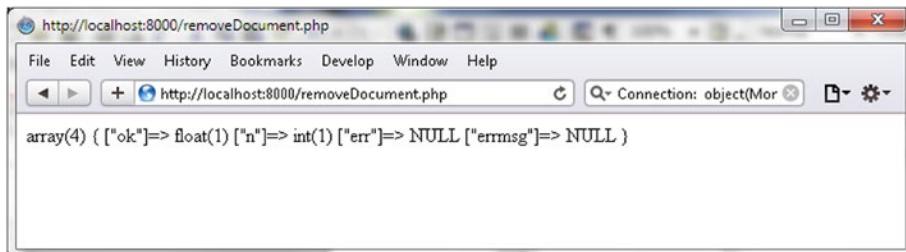
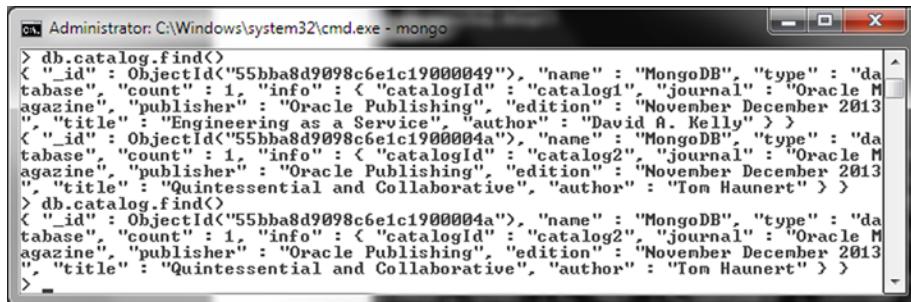


Figure 3-29. Removing a Document

If the `db.catalog.find()` method is run again only one document is listed as shown in the result for the 2nd run of `db.catalog.find()` Figure 3-30.



```
> db.catalog.find()
{
  "_id": ObjectId("55bba8d9098c6e1c19000049"),
  "name": "MongoDB",
  "type": "database",
  "count": 1,
  "info": {
    "catalogId": "catalog1",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
  },
  "title": "Engineering as a Service",
  "author": "David A. Kelly"
}
{
  "_id": ObjectId("55bba8d9098c6e1c1900004a"),
  "name": "MongoDB",
  "type": "database",
  "count": 1,
  "info": {
    "catalogId": "catalog2",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
  },
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}
> db.catalog.find()
{
  "_id": ObjectId("55bba8d9098c6e1c1900004a"),
  "name": "MongoDB",
  "type": "database",
  "count": 1,
  "info": {
    "catalogId": "catalog2",
    "journal": "Oracle Magazine",
    "publisher": "Oracle Publishing",
    "edition": "November December 2013"
  },
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}
>
```

Figure 3-30. Listing a Document after Removing One of the Two Documents

The `remove()` method may be used to remove all documents. As we removed one of the two documents added by running the `addDocumentBatch.php` script we need to add some more documents to the `catalog` collection to demonstrate removing all documents as removing a single document would not demonstrate that all or multiple documents got removed.

1. Run the `addDocumentBatch.php` script again to add two more documents, which brings the total documents to three.
2. Next, create a PHP script `removeAllDocuments.php` in the `C:\php` directory. Invoke the `remove()` method as for removing a single document but supply an empty array.

```
$status=$collection->remove(array());
```

The `removeAllDocuments.php` script is listed.

```
<?php
try
{
$connection = new MongoClient();
$collection=$connection->local->catalog;
$status=$collection->remove(array());
var_dump($status);
}catch (MongoConnectionException $e)
{
    echo '<p>Couldn\'t connect to mongodb</p>';
    exit();
}
?>
```

3. Run the PHP script with the URL `http://localhost:8000/removeAllDocuments.php`. As indicated by `n=>int(3)` in Figure 3-31 all of the three documents get removed.

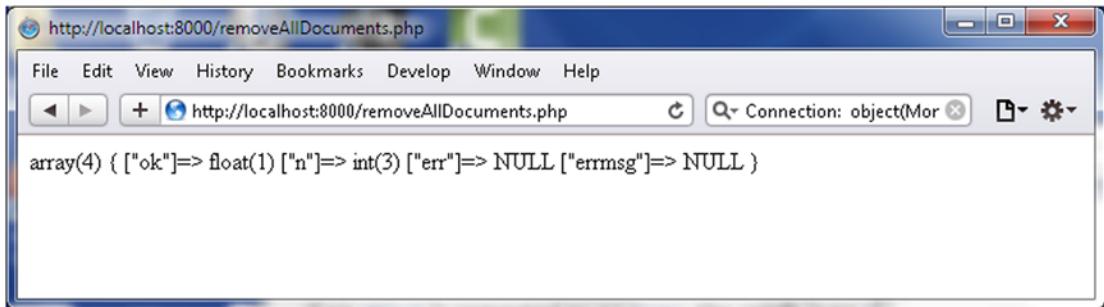


Figure 3-31. Running the *removeAllDocuments.php* Script

The `db.catalog.find()` if run subsequently does not list any documents. Do not remove the `catalog` collection in the local database as we shall be demonstrating dropping a collection in the next section.

Summary

In this chapter we used the PHP Driver for MongoDB to connect to MongoDB and run CRUD operations on the database. We also discussed adding, finding, and updating a single document vs. multiple documents. In the next chapter we shall use Ruby with MongoDB.

CHAPTER 4



Using MongoDB with Ruby

Ruby is an open source programming language, most commonly used in the Ruby on Rails framework. Some of the salient features of Ruby are simplicity, flexibility, extensibility, portability, and OS independent threading. The MongoDB Ruby driver may be used to connect to MongoDB server and add, fetch, and update data in the database. The Ruby driver also provides a C extension for performance. In this chapter we shall discuss using a Ruby client to access and make data changes in MongoDB. This chapter covers the following topics:

- Getting Started
- Using a Collection
- Using Documents

Getting Started

In the following subsections we shall introduce the Ruby Driver for MongoDB, and set up the environment with the required software.

Overview of the Ruby Driver for MongoDB

The Ruby driver for MongoDB API may be used in a Ruby script to connect to MongoDB Server and perform CRUD (create, read, update, and delete) operations on the server. The only namespace in the Ruby driver API is called `Mongo`. The main classes in the `Mongo` namespace are shown in Figure 4-1.

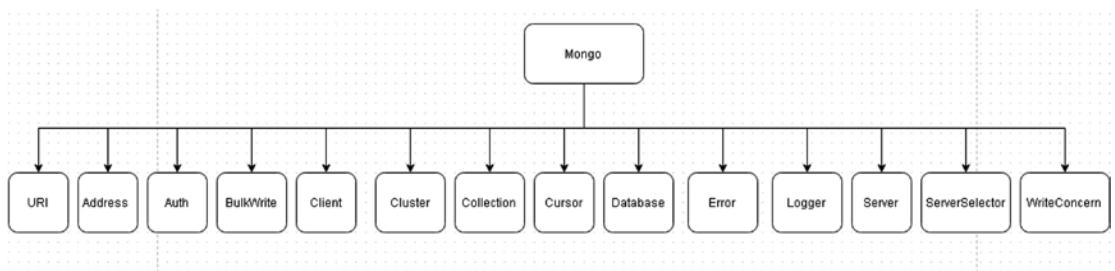


Figure 4-1. Core Classes in Ruby MongoDB Driver

The Mongo namespace classes are discussed in Table 4-1.

Table 4-1. Ruby MongoDB Driver Core Classes

Class	Description	Example Usage
URI	Representation of the MongoDB uri as defined in the connection string format specification.	<code>uri = URI.new('mongodb://localhost:27017')</code>
Address	Represents an address to the server.	<code>Mongo::Address.new("127.0.0.1:27017")</code>
Auth	Represents authentication.	<code>Auth.get(user)</code>
BulkWrite	For bulk write operations.	<code>Mongo::BulkWrite.get(collection, operations, ordered: true)</code>
Client	The Client class is the entry point to the driver.	<code>Mongo::Client.new(['127.0.0.1:27017'])</code>
Cluster	Represents a group of servers.	<code>Mongo::Cluster.new(["127.0.0.1:27017"])</code>
Collection	Represents a collection.	<code>Mongo::Collection.new(database, 'test')</code>
Cursor	Client-side iterator over a query result. Not created directly by a user but created internally by CollectionView.	<code>catalog.find.each { document puts document }</code>
Database	Represents a database on the server.	<code>Mongo::Database.new(client, :test)</code>
Error	Error class.	<code>Mongo::Error::BulkWriteFailure.new(response)</code>
Logger	Logs messages.	<code>Logger.error('mongo', 'message', '10ms')</code>
Server	Represents a single server.	<code>Mongo::Server.new('127.0.0.1:27017' cluster, listeners)</code>
ServerSelector	Given a preference selects a server.	<code>Mongo::ServerSelector.get({ :mode => :secondary })</code>
WriteConcern	Represents write concern.	<code>Mongo::WriteConcern.get(:w => 1)</code>

Setting Up the Environment

We need to download and install the following software to access MongoDB Server from Ruby.

- Ruby Installer for Ruby 2.1.6 (`rubyinstaller-2.1.6-x64.exe`). Download it from <http://rubyinstaller.org/downloads/>. MongoDB Ruby driver 2.x supports Ruby versions 1.8.7, 1.9, 2.0, and 2.1.
- Rubygems.
- RubyInstaller Development Kit (DevKit). Download the appropriate version from <http://rubyinstaller.org/downloads/>. For Ruby 2.1.6 download `DevKit-mingw64-64-4.7.2-20130224-1432-sfx.exe`.
- Ruby driver for MongoDB
- MongoDB Server 3.0.5

Installing Ruby

To install Ruby double-click on the Ruby installer application `rubyinstaller-2.1.6-x64.exe`. The Ruby Setup wizard gets started.

1. Select the Setup Language and click on OK.
2. Accept the License Agreement and click on Next.
3. In Installation Destination and Optional Tasks, specify a destination folder to install Ruby, or select the default folder. The directory path should not include any spaces. Select Add Ruby Executables to your PATH.
4. Click on Install as shown in Figure 4-2.

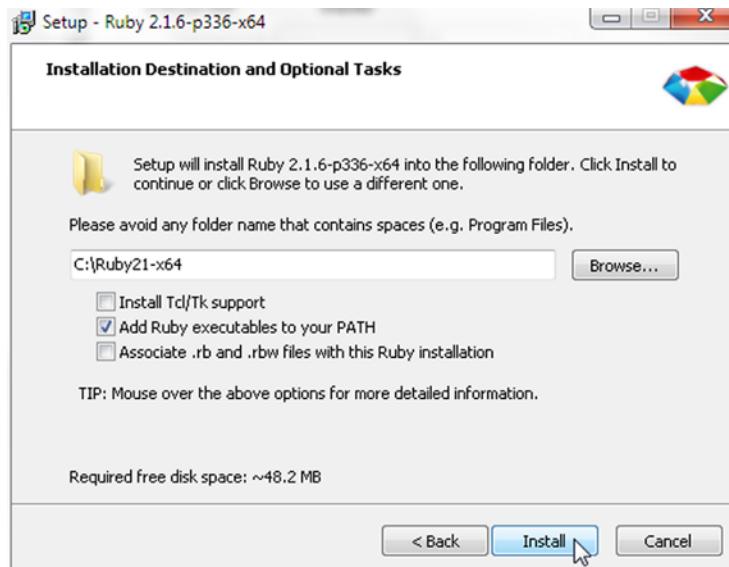


Figure 4-2. Selecting Installation Directory for Ruby

Ruby starts installing as shown in Figure 4-3.

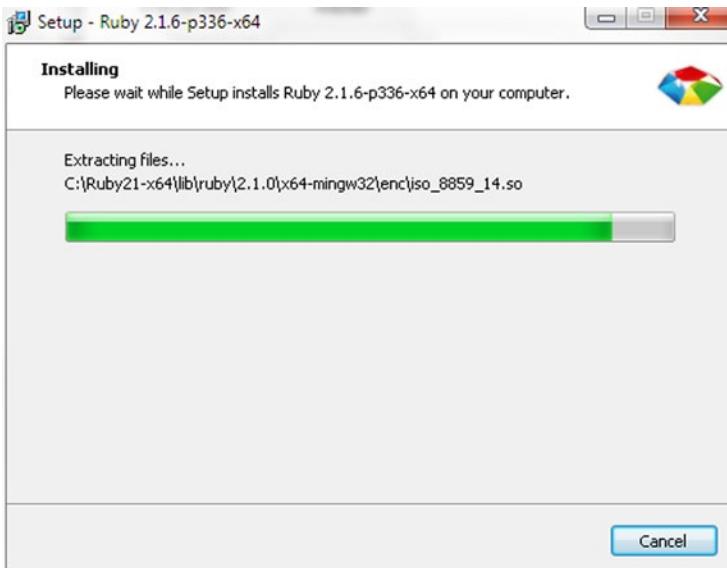


Figure 4-3. Installing Ruby

The Setup wizard completes installing Ruby as shown in Figure 4-4. Click on Finish.



Figure 4-4. Ruby Installed

Next, install Rubygems, which is a package management framework for Ruby. Run the following command to install Rubygems.

```
gem install rubygems-update
```

The Rubygems gem gets installed as shown in Figure 4-5.

```
C:\Ruby21-x64>gem install rubygems-update
Fetching: rubygems-update-2.4.8.gem (100%)
Successfully installed rubygems-update-2.4.8
Parsing documentation for rubygems-update-2.4.8
Installing ri documentation for rubygems-update-2.4.8
Done installing documentation for rubygems-update after 7 seconds
1 gem installed

C:\Ruby21-x64>
```

Figure 4-5. Installing Rubygems

If Rubygems is already installed, update to the latest version with the following command.

```
update_rubygems
```

Rubygems gets updated as shown in Figure 4-6.

```
C:\Ruby21-x64>update_rubygems
RubyGems 2.4.8 installed
Parsing documentation for rubygems-2.4.8
Installing ri documentation for rubygems-2.4.8
```

Figure 4-6. Updating Rubygems

Installing DevKit

DevKit is a toolkit that is used to build many of the C/C++ extensions available for Ruby.

1. To install DevKit double-click on the DevKit installer application (DevKit-mingw64-64-4.7.2-20130224-1432-sfx.exe) to extract DevKit files to a directory. Change directories (cd) to the directory in which the files are extracted.

```
C:\Ruby21-x64
```

2. Initialize DevKit and autogenerate the config.yml file using the following command.

```
ruby dk.rb init
```

3. A config.yml file gets generated in the C:\Ruby21-x64 directory.
Add the following line to the config.yml file.

- C:/Ruby21-x64

4. Install DevKit using the following command.

```
ruby dk.rb install
```

5. Verify that DevKit has been installed using the following command.

```
ruby -rubygems -e "require 'json'; puts JSON.load('[42]').inspect"
```

The output from the preceding commands to initialize/install DevKit are shown in the command shell in Figure 4-7.

```
C:\Ruby21-x64>ruby dk.rb init
Initialization complete! Please review and modify the auto-generated
'config.yml' file to ensure it contains the root directories to all
of the installed Rubies you want enhanced by the DevKit.

C:\Ruby21-x64>ruby dk.rb install
[INFO] Installing 'C:/Ruby21-x64/lib/ruby/site_ruby/2.1.0/rubygems/defaults/operating_system.rb'
[INFO] Installing 'C:/Ruby21-x64/lib/ruby/site_ruby/devkit.rb'

C:\Ruby21-x64>ruby -rubygems -e "require 'json'; puts JSON.load('[42]').inspect"
[42]
C:\Ruby21-x64>
```

Figure 4-7. Installing Devkit

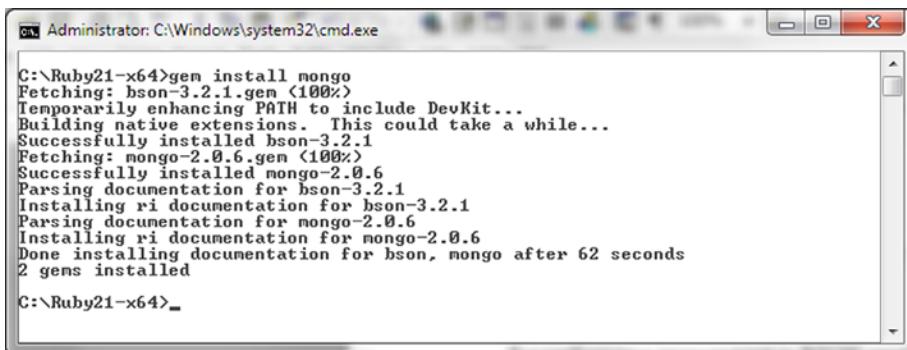
Installing Ruby Driver for MongoDB

To install the MongoDB Ruby driver gem mongo, complete the following steps:

1. Run the following command.

```
>gem install mongo
```

The MongoDB Ruby driver gem gets installed as shown in Figure 4-8.



```
C:\Ruby21-x64>gem install mongo
Fetching: bson-3.2.1.gem (100%)
Temporarily enhancing PATH to include DevKit...
Building native extensions. This could take a while...
Successfully installed bson-3.2.1
Fetching: mongo-2.0.6.gem (100%)
Successfully installed mongo-2.0.6
Parsing documentation for bson-3.2.1
Installing ri documentation for bson-3.2.1
Parsing documentation for mongo-2.0.6
Installing ri documentation for mongo-2.0.6
Done installing documentation for bson, mongo after 62 seconds
2 gems installed

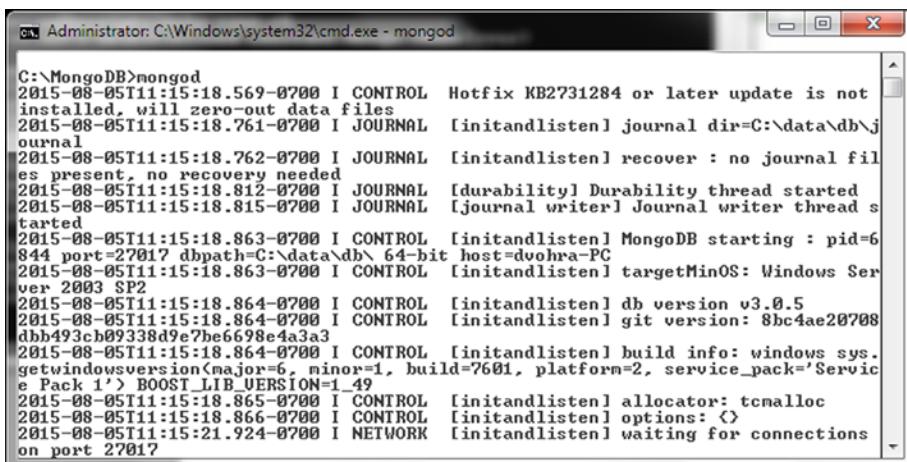
C:\Ruby21-x64>
```

Figure 4-8. Installing MongoDB Ruby Driver

2. Next, install MongoDB Server. Only MongoDB versions 2.4, 2.6, and 3.x are compatible with MongoDB Ruby driver 2.x. Download MongoDB 3.0.5 from www.mongodb.org/ and extract the zip file to a directory.
3. Add the bin directory from the MongoDB installation (for example, C:\Program Files\MongoDB\Server\3.0\bin) to the PATH environment variable.
4. Start MongoDB server with the following command.

```
>mongod
```

MongoDB server gets started as shown in Figure 4-9.



```
C:\MongoDB>mongod
2015-08-05T11:15:18.569-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-08-05T11:15:18.761-0700 I JOURNAL [initandlisten] journal dir=C:\data\db\j
ournal
2015-08-05T11:15:18.762-0700 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-08-05T11:15:18.812-0700 I JOURNAL [durability] Durability thread started
2015-08-05T11:15:18.815-0700 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-08-05T11:15:18.863-0700 I CONTROL [initandlisten] MongoDB starting : pid=6
844 port=27017 dbpath=C:\data\db\ 64-bit host=dvohra-PC
2015-08-05T11:15:18.863-0700 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-08-05T11:15:18.864-0700 I CONTROL [initandlisten] db version v3.0.5
2015-08-05T11:15:18.864-0700 I CONTROL [initandlisten] git version: 8bc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-08-05T11:15:18.864-0700 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-08-05T11:15:18.865-0700 I CONTROL [initandlisten] allocator: tcmalloc
2015-08-05T11:15:18.866-0700 I CONTROL [initandlisten] options: {}
2015-08-05T11:15:21.924-0700 I NETWORK [initandlisten] waiting for connections
on port 27017
```

Figure 4-9. Starting MongoDB Server

Using a Collection

In the following subsections we shall connect to MongoDB server, get database information, and create a collection.

Creating a Connection with MongoDB

In this section we shall connect with MongoDB Server using a Ruby script.

1. Create a directory called C:\Ruby21-x64\mongodbscripts for Ruby scripts.
2. Create a Ruby script connection.rb in the C:\Ruby21-x64\mongodbscripts directory. In the script add a require statement for the mongo gem. Add an include statement for the Mongo namespace.

```
require 'mongo'
include Mongo
```

The Client class provides several attributes and methods for getting information about the connection. Some of the attributes are discussed in Table 4-2.

Table 4-2. Client Class Attributes

Attribute	Description
cluster	The cluster of servers for the client.
database	The database instance.
options	Configuration options.

Some of the salient methods in the Client class are discussed in Table 4-3.

Table 4-3. Client Class Methods

Method	Return type	Description
[]	Mongo::Collection	Gets a collection object.
database_names	Array<String>	Gets names of all databases.
initialize	Client	Instantiates a new driver client.
list_databases	Array<Hash>	Gets info for each database.
read_preference	Object	Gets the read preferences for the provided options.
use(name)	Mongo::Client	Uses the database with the specified name.
with(new_options = {})	Mongo::Client	Provides a new client with the options.
write_concern	Mongo::WriteConcern	Gets the write concern.

The syntax for `Client` class constructor is as follows.

```
initialize(addresses_or_uri, options = {})
```

The parameters for the constructor are discussed in Table 4-4.

Table 4-4. Client Constructor Parameters

Parameter	Type	Description
<code>addresses_or_uri</code>	<code>Array<String>, String</code>	Array of server addresses in the format host:port or a MongoDB URI connection string.
<code>options</code>	<code>Hash</code>	Client options. Defaults to {}.

Some of the options supported by `Client` class constructor are discussed in Table 4-5.

Table 4-5. Client Class Constructor Options

Options	Type	Description
<code>:auth_mech</code>	<code>Symbol</code>	Authentication mechanism to use.
<code>:connect</code>	<code>Symbol</code>	Connection method to use. Value could be <code>:direct</code> , <code>:replica_set</code> , <code>:sharded</code> .
<code>:database</code>	<code>String</code>	The database to connect to.
<code>:user</code>	<code>String</code>	Username.
<code>:password</code>	<code>String</code>	Password.
<code>:max_pool_size</code>	<code>Integer</code>	Maximum size of the connection pool.
<code>:min_pool_size</code>	<code>Integer</code>	Minimum size of the connection pool.
<code>:connect_timeout</code>	<code>Float</code>	Connection timeout in secs.
<code>:read</code>	<code>Hash</code>	Read preference options. The <code>:mode</code> option may be set to <code>:secondary</code> , <code>:secondary_preferred</code> , <code>:primary</code> , <code>:primary_preferred</code> , <code>:nearest</code> .
<code>:replica_set</code>	<code>Symbol</code>	Replica set to connect to.
<code>:write</code>	<code>Hash</code>	Write concern options.

- In the `connection.rb` script create a connection with MongoDB Server using one of the `Client` class constructors. The host and port may be specified explicitly; if either is not specified the default value is used. If the host and port are not specified the default host and port are `localhost:27017`. The default database is `test`. The following are some of the different applications of the `Client` class constructors.

```
client =Mongo::Client.new
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test',
:connect => :direct)
client =Mongo::Client.new('mongodb://127.0.0.1:27017/test')
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test',
:max_pool_size => 20, :connection_timeout => 60)
```

The IPv4 address of the host may be specified instead of localhost.

```
client =Mongo::Client.new(['192.168.1.72:27017'], :database => 'test')
```

The Client class attributes' values may be output, and instance methods may be invoked using the Client class instance.

The connection.rb script for connecting to MongoDB server, invoking some of the methods, and outputting some of the attribute values are listed:

```
require 'mongo'
include Mongo
#client =Mongo::Client.new(['127.0.0.1:27017'], :database => 'test')
#client =Mongo::Client.new(['127.0.0.1:27017'], :database => 'test', :connect => :direct)
#client =Mongo::Client.new('mongodb://127.0.0.1:27017/test')
client = Mongo::Client.new(['127.0.0.1:27017'], :database => 'test', :max_pool_size => 20,
:connection_timeout => 60)
print "Cluster: "
print client.cluster
print "\n"
print "Configuration Options: "
print client.options
print "\n"
print "Read Preference: "
print client.read_preference
print "\n"
print "Write Concern : "
print client.write_concern
print "\n"
```

4. From the C:\Ruby21-x64\mongodbscripts directory run the connection.rb script with the following command.

```
>ruby connection.rb
```

The output from the connection.rb script is shown in Figure 4-10.



The screenshot shows a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe". The command entered is ">ruby connection.rb". The output displays the configuration options for the MongoDB client, including the cluster, configuration options, read preference, and write concern.

```
C:\Ruby21-x64\mongodbscripts>ruby connection.rb
D, [2015-08-05T12:39:28.751637 #156] DEBUG -- : MONGODB : Adding 127.0.0.1:27017
to the cluster. : runtime: 0.0000ms
D, [2015-08-05T12:39:28.775638 #156] DEBUG -- : MONGODB : COMMAND : namespace=ad
min.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil : runtime:
23.0012ms
Cluster: #<Mongo::Cluster:0x00000002d49bd8>
Configuration Options: {:database=>"test", :max_pool_size=>20, :connection_timeo
ut=>60}
Read Preference: #<Mongo::ServerSelector::Primary:0x00000002d38450>
Write Concern : #<Mongo::WriteConcern::Acknowledged:0x00000002d38298>
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-10. Running Ruby Script connection.rb

Connecting to a Database

A MongoDB database instance is represented with the `Mongo::Database` class. In this section we shall create some MongoDB database instances and get other information about databases.

1. Create a Ruby script `db.rb` in the `C:\Ruby21-x64\mongodbscripts` directory. The `Database` class provides several instance attributes and methods. The attributes are discussed in Table 4-6.

Table 4-6. Database Class Instance Attributes

Parameter	Type	Description
<code>client</code>	Client	The database client.
<code>name</code>	String	Database name.
<code>options</code>	Hash	Configuration options.

Some of the instance methods supported by the `Database` class are discussed in Table 4-7.

Table 4-7. Database Class Methods

Method	Return Type	Description
<code>[] or collection</code>	<code>Mongo::Collection</code>	Gets a collection in this database.
<code>collection_names(options = {})</code>	<code>Array<String></code>	Gets all the names of nonsystem collections.
<code>collections</code>	<code>Array<Mongo::Collection></code>	Gets all the collections that belong to this database.
<code>command(operation, opts = {})</code>	<code>Hash</code>	Executes a command on the database.
<code>drop</code>	<code>Result</code>	Drops the database.
<code>initialize(client, name, options = {})</code>	<code>Database</code>	Instantiates a new <code>Database</code> object.
<code>list_collections</code>	<code>Array<Hash></code>	Gets information on all the collections in the database.
<code>users</code>	<code>View::User</code>	Gets the user view for the database.

The `Database` class constructor has the following signature.

```
initialize(client, name, options = {})
```

The constructor parameters are the same as the class attributes and are discussed in Table 4-6.

2. In the db.rb script create a Client instance as before.

```
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
```

3. Obtain the database from the Client instance using the database attribute and output the database name using the name attribute of the database instance.

```
db=client.database
print db.name
```

4. Output the client and the configuration options using the client and options attributes.

```
print db.client
print db.options
```

5. Output the database names using the database_names() method of the Client class.

```
print client.database_names
```

6. Iterate over the collection returned by the list_databases() method of the Client instance to output information about each database.

```
client.list_databases.each { |info| puts info.inspect }
```

7. To use a particular database invoke the use(database) method of the Client instance. For example, the database may be set to mongo database as follows.

```
client=client.use(:mongo)
```

8. Subsequent to setting the database to mongo, output the database name again to verify that the database has been set. Drop a database using the drop() method.

```
print db.drop
```

The db.rb script is listed as follows.

```
require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
print "Database Connected to: "
db=client.database
print db.name
print "\n"
```

```
print "Database Client: "
print db.client
print "\n"

print "Database Options: "
print db.options
print "\n"

print "Database Names: "
print client.database_names
print "\n"

print "Database Info: "
client.list_databases.each { |info| puts info.inspect }
print "\n"

print "Use Database mongo: "
client=client.use(:mongo)
print "\n"

print "Database Connected to: "
db=client.database
print db.name
print "\n"

print "Use Database mongo: "
client=client.use(:local)
print "\n"

print "Database Connected to: "
db=client.database
print db.name
print "\n"

print db.drop
```

9. Run the db.rb script with the following command.

```
>ruby db.rb
```

The output from the db.rb script is shown in Figure 4-11. What should be noted in the output is that the mongo database instance does not get listed with database_names subsequent to setting the database instance to mongo because the database instance has not been accessed.



```
C:\Ruby21-x64\mongodbscripts>ruby db.rb
D, [2015-08-06T14:17:26.152275 #72441 DEBUG -- : MONGODB : Adding 127.0.0.1:2701
7 to the cluster. i runtime: 0.9999ms
D, [2015-08-06T14:17:26.170276 #72441 DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1> flags=[] limit=-1 skip=0 project=nil i runtime
: 18.0013ms
Database Connected to: test
Database Client: #<Mongo::Client:0x000000006b56c0>
Database Options: <:database=>"test">
Database Names: D, [2015-08-06T14:17:26.175276 #72441 DEBUG -- : MONGODB : COMMA
ND : namespace=admin.$cmd selector=<:listDatabases=>1> flags=[{:slave_ok}] limit=-
1 skip=0 project=nil i runtime: 3.0000ms
["Loc8r", "local"]
Database Info: D, [2015-08-06T14:17:26.179276 #72441 DEBUG -- : MONGODB : COMMAN
D : namespace=admin.$cmd selector=<:listDatabases=>1> flags=[{:slave_ok}] limit=-
1 skip=0 project=nil i runtime: 2.0001ms
{"name":>"Loc8r", "sizeOnDisk":>83886080.0, "empty":>false}
{"name":>"local", "sizeOnDisk":>83886080.0, "empty":>false}

Use Database mongo:
Database Connected to: mongo
Database Names: D, [2015-08-06T14:17:26.183276 #72441 DEBUG -- : MONGODB : COMMA
ND : namespace=admin.$cmd selector=<:listDatabases=>1> flags=[{:slave_ok}] limit=-
1 skip=0 project=nil i runtime: 2.0001ms
["Loc8r", "local"]
Use Database mongo:
Database Connected to: local
D, [2015-08-06T14:17:26.317284 #72441 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:dropDatabase=>1> flags=[{:slave_ok}] limit=-1 skip=0 project=
nil i runtime: 132.0071ms
#<Mongo::Operation::Result:0x0000000657958>
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-11. Running Ruby Script db.rb

After a database instance local has been dropped the database does not get listed in Mongo shell with show dbs command as shown in Figure 4-12.



```
> use local
switched to db local
> show dbs
Loc8r 0.078GB
local 0.078GB
> show dbs
Loc8r 0.078GB
```

Figure 4-12. The local database not listed with show dbs

As we shall be using the local database in subsequent sections create the local database with the following commands in Mongo shell.

```
>use local
>db.createCollection("local")
```

Creating a Collection

The `Mongo::Database` class provides several methods for collections, which are discussed in Table 4-7. A collection is represented with the `Mongo::Collection` class. The `Collection` class provides several instance attributes and methods. The attributes are discussed in Table 4-8.

Table 4-8. Collection Class Instance Attributes

Parameter	Type	Description
<code>database</code>	<code>Mongo::Database</code>	The database.
<code>name</code>	<code>String</code>	Collection name.
<code>options</code>	<code>Hash</code>	Configuration options.

Some of the instance methods supported by the `Collection` class are discussed in Table 4-9.

Table 4-9. Collection Class Instance Methods

Method	Return Type	Description
<code>bulk_write(operations, options)</code>	<code> BSON::Document</code>	To run a batch of bulk write operations.
<code>capped?</code>	<code>true, false</code>	Finds if a collection is capped.
<code>create</code>	<code>Result</code>	Creates a collection.
<code>drop</code>	<code>Result</code>	Drops a collection.
<code>find(filter = nil)</code>	<code>CollectionView</code>	Finds documents in a collection.
<code>indexes(options = {})</code>	<code>View::Index</code>	Returns a view of all indexes of the collection.
<code>insert_many(documents, options = {})</code>	<code>Result</code>	Inserts the provided documents in the collection.
<code>insert_one(document, options = {})</code>	<code>Result</code>	Inserts a single document in the collection.
<code>namespace</code>	<code>String</code>	Gets a fully qualified namespace of the collection.

The `Collection` class constructor has the following signature.

```
initialize(database, name, options = {})
```

The constructor parameters are the same as the class attributes.

1. First, create a Ruby script `collection.rb` in the `C:\Ruby21-x64\mongodbscripts` directory and include the `mongo` gem and the `Mongo` namespace as before.
2. Create a `Client` instance for a connection to MongoDB and subsequently get the database instance for local database.

```
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
```

3. Using the `use(database)` instance method in `Client` class. set the database to local.

```
client=client.use(:local)
```

4. Output the collection names using the `collection_names()` method in `Mongo::Database` class.

```
print db.collection_names({})
```

5. Output info on all the collections using the `list_collections()` method.

```
db.list_collections.each { |info| puts info.inspect }
```

6. Output the array of all the collections using the `collections()` method.

```
db.collections.each { |info| puts info.inspect }
```

7. Using the `Database` class instance method `[]`, set the collection to `users` as follows.

```
collection=db[:users]
```

8. Output the collection name using the `name` attribute in `Collection` class.

```
print collection.name
```

9. The database associated with a collection may be output using the `database` attribute.

```
print collection.database
```

10. Find if a collection is capped using the `capped?` method.

```
print collection.capped?
```

11. Output the namespace using the `namespace()` method.

```
print collection.namespace
```

12. The `collection()` instance method in `Database` class does not create a collection until the collection is accessed. To demonstrate, invoke the `collection()` method on a collection that does not exist in the database, for example, the `mongodb` collection.

```
collection=db.collection("mongodb")
```

13. Subsequently, output the collection names.

```
print db.collection_names({})
```

When the script is run the mongodb collection does not get listed as shown in Figure 4-13.

```
C:\>Ruby21-x64\mongodbscripts>ruby collection.rb
D, [2015-08-05T13:41:58.189092 #6012] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
7 to the cluster. | runtime: 0.000ms
D, [2015-08-05T13:41:58.212094 #6012] DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil | runtime
: 22.0010ms
Use Database local:
Database Connected to: local
Collection Names: D, [2015-08-05T13:41:58.219094 #6012] DEBUG -- : MONGODB : COM
MAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
<:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=nil
| runtime: 3.000ms
["locations", "users"]
Collections List: D, [2015-08-05T13:41:58.223094 #6012] DEBUG -- : MONGODB : COM
MAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
<:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=nil
| runtime: 2.001ms
<"name">>"locations", "options"=>>>
<"name">>"users", "options"=>>>

Collections Array: D, [2015-08-05T13:41:58.227095 #6012] DEBUG -- : MONGODB : CO
MMAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
=><:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=n
il | runtime: 0.9999ms
#<Mongo::Collection:0x2302300 namespace=local.locations>
#<Mongo::Collection:0x2302180 namespace=local.users>

Collection name: users
Collection Database: #<Mongo::Database:0x000000004ac838>
Collection options: {}
Capped: D, [2015-08-05T13:41:58.248096 #6012] DEBUG -- : MONGODB : COMMAND : nam
espace=local.$cmd selector=<:collstats=>"users" flags=[:slave_ok] limit=-1 skip
=0 project=nil | runtime: 14.0009ms
false
Namespace: local.users
Collection Names: D, [2015-08-05T13:41:58.252096 #6012] DEBUG -- : MONGODB : COM
MAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
<:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=nil
| runtime: 1.9999ms
["locations", "users"]
D, [2015-08-05T13:41:58.263097 #6012] DEBUG -- : MONGODB : COMMAND : namespace=1
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil | runtime: 10.0012ms
Collection Names: D, [2015-08-05T13:41:58.266097 #6012] DEBUG -- : MONGODB : COM
MAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
<:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=nil
| runtime: 1.9999ms
["locations", "mongodb", "users"]
Drop collection mongodb
D, [2015-08-05T13:41:58.269097 #6012] DEBUG -- : MONGODB : COMMAND : namespace=1
ocal.$cmd selector=<:drop=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 project=
nil | runtime: 0.9999ms
Collection Names: D, [2015-08-05T13:41:58.272097 #6012] DEBUG -- : MONGODB : COM
MAND : namespace=local.$cmd selector=<:listCollections=>1, :cursor=>{}, :filter=
<:name=><"$not">/system\.!$/>>> flags=[:slave_ok] limit=-1 skip=0 project=nil
| runtime: 2.0003ms
["locations", "users"]
```

Figure 4-13. Running the collection.rb script

14. To create the mongodb collection, invoke the `create()` method.

```
collection.create
```

15. Invoke the `collection_names({})` method in `Database` again. As the `create()` method creates the collection the `mongodb` collection gets listed when the `collection.rb` script is run as shown in Figure 4-13.

16. To drop the current collection, invoke the `drop()` method.

```
collection.drop
```

17. List the collection names subsequent to dropping the collection. When the `collection.rb` script is run the `mongodb` collection does not get listed as shown in Figure 4-13. The `collection.rb` script is listed:

```
require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')

print "Use Database local: "
client=client.use(:local)
print "\n"

print "Database Connected to: "
db=client.database
print db.name
print "\n"

print "Collection Names: "
print db.collection_names({})
print "\n"

print "Collections List: "
db.list_collections.each { |info| puts info.inspect }
print "\n"

print "Collections Array: "
db.collections.each { |info| puts info.inspect }
print "\n"

collection=db[:users]

print "Collection name: "
print collection.name
print "\n"

print "Collection Database: "
print collection.database
print "\n"

print "Collection options: "
print collection.options
print "\n"

print "Capped: "
print collection.capped?
print "\n"

print "Namespace: "
print collection.namespace
print "\n"
```

```

## Does not create a collection till the collection is accessed.
collection=db.collection("mongodb")
print "Collection Names: "
print db.collection_names({})
print "\n"

collection.create
print "Collection Names: "
print db.collection_names({})
print "\n"

print "Drop collection mongodb "
print "\n"
collection.drop
print "Collection Names: "
print db.collection_names
print "\n"

```

18. Run the `collection.rb` script with the following command.

```
>ruby collection.rb
```

The output from the `collection.rb` script is shown in Figure 4-13.

Using Documents

In the following subsections we shall add a document to MongoDB server, add a batch of documents, find a single document, find multiple documents, update documents, delete documents, and perform bulk operations.

Adding a Document

In this section we shall add a single document to a MongoDB collection. The `insert_one(document, options = {})` method in the `Collection` class is used to add a document. The `document` parameter is the document to add. The `options` parameter is a customization Hash of options that defaults to `{}`.

1. Create a Ruby script `addDocument.rb` and add `require` and `include` statements for `mongo` gem and Mongo namespace respectively.
2. Create a connection, set database to `local` and get the `mongodb` collection.

```

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

```

3. The `create()` method must be invoked as the `mongodb` collection if it does not already exist.

```
collection.create
```

4. Create a document JSON to add.

```
document1={
  "_id" => "document1a",
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}
```

5. Invoke the `insert_one()` method to add the document to the `mongodb` collection.

```
collection.insert_one(document1)
```

6. Similarly add another document.

```
document2={
  "_id" => "document1a",
  "catalogId" => "catalog2",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}
collection.insert_one(document2)
```

The `addDocument.rb` script is listed:

```
require 'mongo'
include Mongo
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

collection.create

document1={
  "_id" => "document1a",
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}
```

```

collection.insert_one(document1)

document2={
    "catalogId" => "catalog2",
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Quintessential and Collaborative",
    "author" => "Tom Haunert"
}

collection.insert_one(document2)

```

7. Run the `addDocument.rb` script with the following command.

```
>ruby addDocument.rb
```

The output from the `addDocument.rb` script is shown in Figure 4-14.

```

Administrator: C:\Windows\system32\cmd.exe
C:\Ruby21-x64\mongodbscripts>ruby addDocument.rb
D, [2015-08-05T14:00:53.727042 #6492] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. : runtime: 0.000ms
D, [2015-08-05T14:00:53.741042 #6492] DEBUG -- : MONGODB : COMMAND : namespace=a
dmn.$cmd selector=<:ismaster=>1> flags=[l limit=-1 skip=0 project=nil : runtime
: 13.000ms
D, [2015-08-05T14:00:53.744042 #6492] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil : runtime: 1.9999ms
D, [2015-08-05T14:00:53.746043 #6492] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":>"catalog1",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novem
ber December 2013", "title":>"Engineering as a Service", "author":>"David A. Kel
ly", "_id":>BSON::ObjectId('55c... flags=[l limit=-1 skip=0 project=nil : runtime
: 0.9999ms
D, [2015-08-05T14:00:53.747043 #6492] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"_id":>"document1", "cata
logId":>"catalog2", "journal":>"Oracle Magazine", "publisher":>"Oracle Publishin
g", "edition":>"November December 2013", "title":>"Quintessential and Collaborat
ive", "author":>"Tom Haunert"}... flags=[l limit=-1 skip=0 project=nil : runtime
: 0.000ms
C:\Ruby21-x64\mongodbscripts>

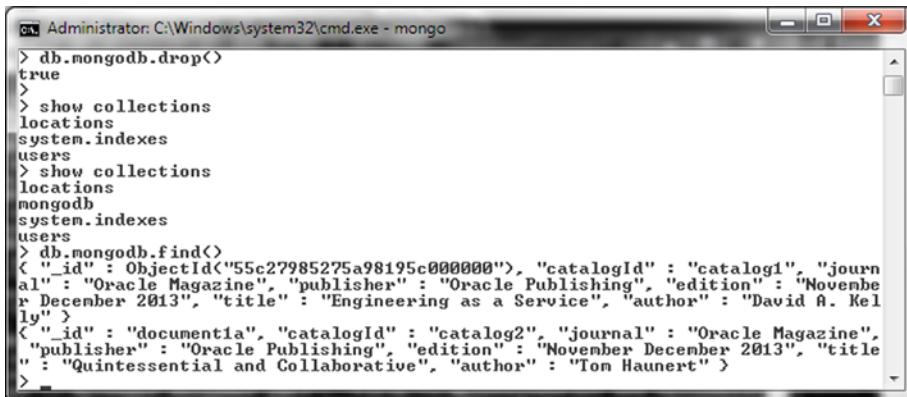
```

Figure 4-14. Running the `addDocument.rb` script

8. Run the following JavaScript method in Mongo shell.

```
>db.mongodbfind()
```

The two documents added get listed as shown in Figure 4-15.



```

Administrator: C:\Windows\system32\cmd.exe - mongo
> db.mongodb.drop()
true
>
> show collections
locations
system.indexes
users
> show collections
locations
mongodb
system.indexes
users
> db.mongodb.find()
{
  "_id": ObjectId("55c27985275a98195c000000"),
  "catalogId": "catalog1",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "November December 2013",
  "title": "Engineering as a Service",
  "author": "David A. Kelly"
}
{
  "_id": "document1a",
  "catalogId": "catalog2",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "November December 2013",
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}

```

Figure 4-15. Finding and listing Documents added

- The _ids of the documents added must be unique or the OperationFailure error gets generated. To demonstrate add two documents with the same _id. The Ruby script addDocument.rb to demonstrate OperationFailure error is listed:

```

require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

collection.create

document1={
  "_id" => "document1a",
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}

collection.insert_one(document1)

document2={
  "_id" => "document1a",
  "catalogId" => "catalog2",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}
collection.insert_one(document2)

```

When the script is run again with ruby addDocument.rb, the OperationFailure error gets generated as shown in Figure 4-16.

```
C:\Ruby21-x64\mongodbscripts>ruby addDocument.rb
D, [2015-08-05T14:05:28.724778 #29321 DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. i runtime: 0.999ms
D, [2015-08-05T14:05:28.736771 #29321 DEBUG -- : MONGODB : COMMAND : namespace=a
dmn.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil i runtime
: 12.0010ms
D, [2015-08-05T14:05:28.740771 #29321 DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil i runtime: 3.0000ms
D, [2015-08-05T14:05:28.741771 #29321 DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:insert=>"mongodb", :documents=>[{"_id":>"documentia", "cata
logId":>"catalog1", "journal":>"Oracle Magazine", "publisher":>"Oracle Publishin
g", "edition":>"November December 2013", "title":>"Engineering as a Service", "a
uthor":>"David A. Kelly"], :w... flags=[] limit=-1 skip=0 project=nil i runtime
: 0.999ms
D, [2015-08-05T14:05:28.802775 #29321 DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:insert=>"mongodb", :documents=>[{"_id":>"documentia", "cata
logId":>"catalog2", "journal":>"Oracle Magazine", "publisher":>"Oracle Publishin
g", "edition":>"November December 2013", "title":>"Quintessential and Collaborat
ive", "author":>"Tom Haunert"}... flags=[] limit=-1 skip=0 project=nil i runtime
: 60.003ms
C:/Ruby21-x64/lib/ruby/gems/2.1.0/gems/mongo-2.0.6/lib/mongo/operation/result.rb
:214:in `validate!': E11000 duplicate key error index: local.mongodb._id_ dup k
ey: < : "documentia" > (11000) (Mongo::Error::OperationFailure)
  from C:/Ruby21-x64/lib/ruby/gems/2.1.0/gems/mongo-2.0.6/lib/mongo/operat
ion/write/insert.rb:72:in `execute_write_command'
  from C:/Ruby21-x64/lib/ruby/gems/2.1.0/gems/mongo-2.0.6/lib/mongo/operat
ion/write/insert.rb:62:in `execute'
  from C:/Ruby21-x64/lib/ruby/gems/2.1.0/gems/mongo-2.0.6/lib/mongo/operat
ion/write/insert_many.rb:196:in `insert_many'
  from C:/Ruby21-x64/lib/ruby/gems/2.1.0/gems/mongo-2.0.6/lib/mongo/operat
ion/write/insert_one.rb:175:in `insert_one'
  from addDocument.rb:38:in `<main>'

C:\Ruby21-x64\mongodbscripts>_
```

Figure 4-16. OperationFailure Error due to Duplicate Key

Adding Multiple Documents

In the preceding section we added two documents but by invoking the `insert_one()` method twice. The `Mongo::Collection` class provides the `insert_many(documents, options = {})` method to add multiple documents.

1. Create a Ruby script `addDocuments.rb` in the `C:\Ruby21-x64\mongodbscripts` directory.
2. Create a `Client` instance and get the `mongodb` collection as before. So that the newly added documents do not have duplicate key with a document already in the `mongodb` collection, delete the `mongodb` collection before running the `addDocuments.rb` script using the `db.mongodb.drop()` method in the `mongo` shell.

```
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")
collection.create
```

3. Create JSON for two documents.

```
document1={
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}

document2={

  "catalogId" => "catalog2",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}
```

4. Invoke the `insert_many()` method to add the two documents.

```
collection.insert_many([document1,document2])
```

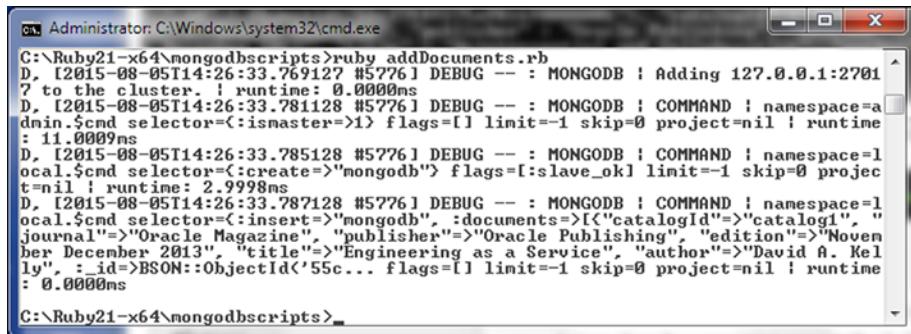
The `addDocuments.rb` is listed:

```
require 'mongo'
include Mongo
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")
collection.create
document1={
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}

document2={

  "catalogId" => "catalog2",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}
collection.insert_many([document1,document2])
```

Run the script with `ruby addDocuments.rb`. The output from the `addDocuments.rb` script is shown in Figure 4-17.



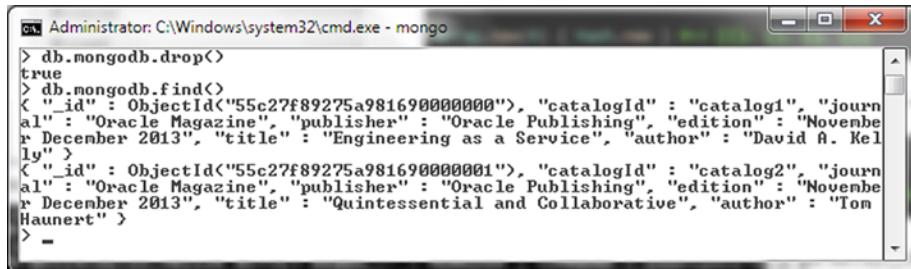
```
C:\Ruby21-x64\mongodbscripts>ruby addDocuments.rb
D, [2015-08-05T14:26:33.769127 #5776] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. | runtime: 0.0000ms
D, [2015-08-05T14:26:33.781128 #5776] DEBUG -- : MONGODB : COMMAND : namespace=admin.$cmd selector=<{:ismaster=>1} flags=[] limit=-1 skip=0 project=nil | runtime: 11.009ms
D, [2015-08-05T14:26:33.785128 #5776] DEBUG -- : MONGODB : COMMAND : namespace=admin.$cmd selector=<{:create=>"mongodb"} flags=[{:slave_ok}] limit=-1 skip=0 project=nil | runtime: 2.9998ms
D, [2015-08-05T14:26:33.787128 #5776] DEBUG -- : MONGODB : COMMAND : namespace=admin.$cmd selector=<{:insert=>"mongodb", :documents=>[{"catalogId":>"catalog1", "journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November December 2013", "title":>"Engineering as a Service", "author":>"David A. Keil", "_id=>BSON::ObjectId<'55c... flags=[] limit=-1 skip=0 project=nil | runtime: 0.0000ms
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-17. Output from `addDocuments.rb` Script

- To verify that the two documents got added, run the following command in Mongo shell.

```
>db.mongodb.find()
```

The two documents added get listed as shown in Figure 4-18.



```
C:\Administrator: C:\Windows\system32\cmd.exe - mongo
> db.mongodb.drop()
true
> db.mongodb.find()
{ "_id" : ObjectId("55c27f89275a981690000000"), "catalogId" : "catalog1", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Engineering as a Service", "author" : "David A. Keil" }
{ "_id" : ObjectId("55c27f89275a981690000001"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" }
> -
```

Figure 4-18. Listing the Documents Added with `insert_many`

In this section we shall also add multiple documents using a `for` loop.

- Drop the `mongodb` collection with `db.mongodb.drop()`.
- Create another Ruby script `addDocument2.rb` and create the collection `mongodb` in the local database. First get the local database instance and subsequently invoke the `create()` method to create a collection.

```
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")
collection.create
```

3. Using a `for` loop, create a variable for `catalogId` and create a new document instance using the `catalogId` variable value in each iteration of the `for` loop. Add the document instance to the collection using the `insert_one()` method. The `addDocument2.rb` script is listed:

```
require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")
collection.create
for i in 1..10 do
  catalogId="catalog"+i.to_s
  document={
    "catalogId" => catalogId,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing"
  }
  collection.insert_one(document)
  print "\n"
end
```

4. Run the `addDocument2.rb` script with the following command.

```
>ruby addDocument2.rb
```

Multiple documents get added as shown by the output from the script in Figure 4-19.

```
C:\Ruby21-x64\mongodbscripts>ruby addDocument2.rb
D, [2015-08-05T14:14:59.150397 #4408] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. | runtime: 0.000ms
D, [2015-08-05T14:14:59.165398 #4408] DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=1> flags=[] limit=-1 skip=0 project=nil | runtime
: 15.0013ms
D, [2015-08-05T14:14:59.169398 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil | runtime: 3.000ms
D, [2015-08-05T14:14:59.170398 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog1",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000001')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 0.9999ms
D, [2015-08-05T14:14:59.173398 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog2",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000002')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 0.9999ms
D, [2015-08-05T14:14:59.176398 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog3",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000003')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 1.0002ms
D, [2015-08-05T14:14:59.179399 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog4",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000004')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 0.0000ms
D, [2015-08-05T14:14:59.182399 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog5",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000005')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 1.0002ms
D, [2015-08-05T14:14:59.184399 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog6",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000006')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 0.0000ms
D, [2015-08-05T14:14:59.186399 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog7",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000007')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 0.9999ms
D, [2015-08-05T14:14:59.188399 #4408] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog8",
"journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", :_id:>BSON::Objec
tId('55c27cd3275a981138000008')}, :writeConcern=>{:w=>1}, :ordered=>true} flags
=[] limit=-1 skip=0 project=nil | runtime: 1.0002ms
```

Figure 4-19. Running the addDocument2.rb Script

- Subsequently run the following JavaScript method in Mongo shell to list the documents added.

```
>db.mongodb.find()
```

The documents added get listed as shown in Figure 4-20.

```
> db.mongodb.find()
{
  "_id": ObjectId("55c27cd3275a981138000000"),
  "catalogId": "catalog1",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000001"),
  "catalogId": "catalog2",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000002"),
  "catalogId": "catalog3",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000003"),
  "catalogId": "catalog4",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000004"),
  "catalogId": "catalog5",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000005"),
  "catalogId": "catalog6",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000006"),
  "catalogId": "catalog7",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000007"),
  "catalogId": "catalog8",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000008"),
  "catalogId": "catalog9",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
},
{
  "_id": ObjectId("55c27cd3275a981138000009"),
  "catalogId": "catalog10",
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing"
}

```

Figure 4-20. Listing Multiple Documents Added with a for Loop

Finding a Single Document

The `find(filter = nil)` method in `Mongo::Collection` class is used to find one or more documents. By default, if no filter is specified, all documents get found. In this section we shall find a single document using a filter.

1. Create a Ruby script `findDocument.rb` in the `C:\Ruby21-x64\mongodbscripts` directory.
2. Create the `mongodb` collection as before.
3. Add two documents using the `insert_many()` method as in the `addDocuments.rb` script.
4. Invoke the `find()` method with a filter specifying `catalogId` as `catalog1`. Iterate over the result cursor to output the documents found.

```
collection.find(:catalogId=>"catalog1").each do |document|
  print document
end
```

The `findDocument.rb` script is listed below.

```
require 'mongo'
include Mongo
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

collection.create
document1= {
  "catalogId" => "catalog1",
  "journal" => "Oracle Magazine",
```

```

"publisher" => "Oracle Publishing",
"edition" => "November December 2013",
"title" => "Engineering as a Service",
"author" => "David A. Kelly"
}

document2= {
  "catalogId" => "catalog2",
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}

collection.insert_many([document1,document2])

collection.find(:catalogId=>"catalog1").each do |document| puts document
end

```

5. Drop the mongodb collection with db.mongodb.drop(). Run the script with the following command.

```
>ruby findDocument.rb
```

The document with catalogId catalog1 gets found and output as shown in Figure 4-21.

```

Administrator: C:\Windows\system32\cmd.exe
C:\Ruby21-x64\mongodbscripts>ruby findDocument.rb
D, [2015-08-06T16:37:00.743274 #7316] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. ! runtime: 0.0000ms
D, [2015-08-06T16:37:00.758275 #7316] DEBUG -- : MONGODB : COMMAND : namespace=a
dmn.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil ! runtime
: 14.001ms
D, [2015-08-06T16:37:00.762275 #7316] DEBUG -- : MONGODB : COMMAND : namespace=lo
cal.$cmd selector=<:create=>"mongodb" flags=[{:slave_ok}] limit=-1 skip=0 projec
t=nil ! runtime: 3.0000ms
D, [2015-08-06T16:37:00.763275 #7316] DEBUG -- : MONGODB : COMMAND : namespace=lo
cal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":>"catalog1", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novem
ber December 2013", "title":>"Engineering as a Service", "author":>"David A. Kel
ly", "_id":> BSON::ObjectId<'55c3ef9c275a981c94000000' }, {"catalogId":>"catalog1", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novembe
r December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly
"}]
C:\Ruby21-x64\mongodbscripts>

```

Figure 4-21. Finding a Single Document

Finding Multiple Documents

In this section we shall find multiple documents using the `find()` method. The `find(filter = nil)` method may be used to find multiple documents that match the filter. The `find()` method returns a cursor. If the `find()` method invocation includes a block, as with invoking the `each()` method on the result of invoking the `find()` method, the method yields a cursor to the block that may be iterated over to output the documents.

1. To find documents using the `find()` method create a Ruby script `findDocuments.rb` in the `C:\Ruby21-x64\mongodbscripts` directory.
2. Create a collection instance as discussed before. The `find()` method by itself does not return a `Cursor` object but returns a `CollectionView` that creates a `Cursor`. Each of the following returns a `CollectionView` object.

```
print collection.find()
print collection.find({"journal" => "Oracle Magazine"})
```

3. The `Cursor` class implements the `Enumerable`, which implies that the `Enumerable` methods such as `each` may be used. To output the documents in the `Cursor`, iterate over the result set using the `each` method.

```
collection.find.each { |document| puts document }
```

4. To output all documents use the `Enumerable#find.to_a()` method, which gets all documents into the memory at once thus making the method inefficient in comparison to the `each()` method with which one document at a time is processed in the block iteration.

```
puts collection.find.to_a
```

5. In the `findDocuments.rb` script create an array of documents and add using the `insert_many` script.

```
collection.insert_many([document1,document2])
```

6. Try each of the following options for finding documents:

- Using the `find()` method find all documents and invoke `each` over the `Cursor` generated by the `CollectionView` to iterate over the collection of documents returned.

```
collection.find().each do |document|
  print document
end
```

- As another example find only documents with `:journal` set to 'Oracle Magazine' and limit the number of documents returned using the `limit()` method.

```
collection.find(:journal => 'Oracle Magazine').limit(5).each do |document|
  print document
end
```

- To find the number of documents for a particular edition use the `count()` method.

```
print collection.find(:edition => 'November December 2013').count
```

- Distinct documents may be found using the `distinct()` method.

```
print collection.find.distinct(:catalogId)
collection.find(:journal => 'Oracle Magazine').limit(5).each do |document|
    print document
end
```

The `findDocuments.rb` script is listed below.

```
require 'mongo'
include Mongo
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")
collection.create
document1={
    "catalogId" => "catalog1",
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Engineering as a Service",
    "author" => "David A. Kelly"
}
document2={
    "catalogId" => "catalog2",
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Quintessential and Collaborative",
    "author" => "Tom Haunert"
}
collection.insert_many([document1,document2])
collection.find().each do |document|
    print document
end
collection.find(:journal => 'Oracle Magazine').limit(5).each do |document|
    print document
end
print "Number of documents with edition November December 2013: "
print "\n"
print collection.find(:edition => 'November December 2013').count
print "\n"
print "Number of distinct documents by catalogId: "
print "\n"
print collection.find.distinct(:catalogId)
```

7. Drop the `mongodb` collection with `db.mongodb.drop()` as the collection is used again. Run the `findDocuments.rb` script with the following command.

```
>ruby findDocuments.rb
```

The output from the `findDocuments.rb` script is shown in Figure 4-22.

```
C:\Ruby21-x64\mongodbscripts>ruby findDocuments.rb
D, [2015-08-05T15:11:39.152866 #36321 DEBUG -- : MONGODB : Adding 127.0.0.1:2701
7 to the cluster. 1 runtime: 0.0000ms
D, [2015-08-05T15:11:39.165867 #36321 DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1 flags=[1 limit=-1 skip=0 project=nil 1 runtime
: 13.0010ns
D, [2015-08-05T15:11:39.169867 #36321 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[{:slave_okl} limit=-1 skip=0 projec
t=nil 1 runtime: 2.0001ms
D, [2015-08-05T15:11:39.170867 #36321 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":>"catalog1", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novem
ber December 2013", "title":>"Engineering as a Service", "author":>"David A. Kel
ly", :_id=>BSON::ObjectId('55c... flags=[1 limit=-1 skip=0 project=nil 1 runtime
: 0.9999ms
D, [2015-08-05T15:11:39.172867 #36321 DEBUG -- : MONGODB : QUERY : namespace=loc
al.mongodb selector=<> flags=[{:slave_okl} limit=0 skip=0 project=nil 1 runtime: 1
.0002ms
<"_id":>BSON::ObjectId('55c28a1b275a980e30000000'), "catalogId":>"catalog1", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novembe
r December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly
"><"_id":>BSON::ObjectId('55c28a1b275a980e30000001'), "catalogId":>"catalog2", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novem
ber December 2013", "title":>"Quintessential and Collaborative", "author":>"Tom
Haunert">D, [2015-08-05T15:11:39.175867 #36321 DEBUG -- : MONGODB : QUERY : name
space=local.mongodb selector=<:journal=>"Oracle Magazine" flags=[{:slave_okl} lim
it=5 skip=0 project=nil 1 runtime: 0.9999ms
<"_id":>BSON::ObjectId('55c28a1b275a980e30000000'), "catalogId":>"catalog1", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novembe
r December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly
"><"_id":>BSON::ObjectId('55c28a1b275a980e30000001'), "catalogId":>"catalog2", "j
ournal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"Novem
ber December 2013", "title":>"Quintessential and Collaborative", "author":>"Tom
Haunert">Number of documents with edition November December 2013:
D, [2015-08-05T15:11:39.181868 #36321 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:count=>"mongodb", :query=><:edition=>"November December 201
3"> flags=[{:slave_okl} limit=-1 skip=0 project=nil 1 runtime: 0.0000ms
2
Number of distinct documents by catalogId:
D, [2015-08-05T15:11:39.184868 #36321 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:distinct=>"mongodb", :key=>"catalogId", :query=><> flags=[{:s
lave_okl} limit=-1 skip=0 project=nil 1 runtime: 1.0002ms
["catalog1", "catalog2"]
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-22. Finding Multiple Documents

The output from `collection.find()` lists the two documents in the `mongodb` collection. The output from `collection.find(:journal => 'Oracle Magazine').limit(5)` outputs documents with `:journal` set to 'Oracle Magazine'. The `limit()` method limits the result to 5, but because the collection has only 2 documents the `limit` method does not have an effect. The `collection.find(:edition => 'November December 2013').count()` method invocation outputs the document count for documents with edition as 'November December 2013' as 2. The `collection.find.distinct(:catalogId)` method finds the 2 distinct `catalogIds` in the `mongodb` collection.

The `find()` method returns a collection view and when each is invoked on the view a `Cursor` object is created. The `Mongo::Collection::View` is a representation of a query and options generating a result set of documents. The `find()` method by itself does not send the query to the server. Another method has to be invoked subsequent to the `find()` method to send the query to the server. For example, when the `each()` method is invoked on a View a `Cursor` object is created which sends the query to the server. The `Mongo::Collection::View` class supports the methods discussed in Table 4-10. These methods are included from `Writable`, `Explainable`, `Readable`, and `Iterable` classes.

Table 4-10. Mongo::Collection::View Class Instance Methods

Method	Return Type	Description
delete_many	Result	Deletes multiple documents.
delete_one	Result	Deletes one document.
find_one_and_delete	BSON::Document	Finds and deletes a document.
find_one_and_replace(replacement, opts = {})	BSON::Document	Finds and replaces a document.
find_one_and_update(document, opts = {})	BSON::Document	Finds and updates a document.
replace_one(document, opts = {})	Result	Replaces a single document.
update_many(spec, opts = {})	Result	Updates multiple documents.
update_one(spec, opts = {})	Result	Updates a single document.
explain	Hash	Gets the explain plan for the query.
aggregate(pipeline, options = {})	Aggregation	Creates an aggregation of the collection view.
allow_partial_results	View	Allows the query to get partial results if some shards are down.
batch_size(batch_size = nil)	(Integer, View)	Sets the batch size. Setting to 1 or a -ve value is equivalent to setting a limit.
comment(comment = nil)	(String, View)	Associates a comment with the query.
count(options = {})	Integer	Gets a count of matching documents.
distinct(field_name, options = {})	Array<Object>	Gets a list of distinct values.
hint(hint = nil)	Hash, View	The index MongoDB has to use on a query.
limit(limit = nil)	Integer, View	Limits the maximum number of documents to return from the query.
map_reduce(map, reduce, options = {}))	MapReduce	Runs a map reduce function on the query.
max_scan(value = nil)	Integer, View	Sets the maximum number of documents to scan.
no_cursor_timeout	View	By default the server times out idle cursors after 10 minutes to prevent excessive memory use. Setting no_cursor_timeout does not time out idle cursors.
projection(document = nil)	Hash, View	The fields to include or exclude from each document in the result set.
read(value = nil)	Symbol, View	The read preference.
show_disk_loc(value = nil)	(true, false, View)	Sets whether disk location should be shown for each document. Value may be true, false, or nil (the default).

(continued)

Table 4-10. (continued)

Method	Return Type	Description
skip(number = nil)	Integer, View	Number of documents to skip. Set to an Integer value.
snapshot(value = nil)	Object	When set to true prevents documents from returning more than once. Value may be true, false, or nil.
sort(spec = nil)	Hash, View	The key and direction pairs specified as Hash by which the result set is sorted.
each { Each ... }	Enumerator	Used to iterate through the documents in the result set.

Updating Documents

The Collection class does not directly provide any instance methods for updating documents. But the collection view returned by the `find()` method provides several methods as discussed in Table 4-10 in the preceding section to update or replace document/s. In this section we shall update documents.

1. Create a Ruby script `updateDocument.rb` in the `C:\Ruby21-x64\mongodbscripts` directory.
2. Delete the `mongodb` collection if already present and create the `mongodb` collection in the `updateDocument.rb` script.
3. Add two documents to the `mongodb` collection.
4. Create a collection view for the `mongodb` collection.

```
collection = client[:mongodb]
```

We shall discuss several examples for which the two documents shall be added to the `mongodb` collection and subsequently updated or replaced.

Example 1

1. For the first example, use the `update_one()` method to increment the `catalogId` of one of the documents by 1. The `$inc` update operator is used to increment the `catalogId`. The `find()` method finds all documents with `journal` set to 'Oracle Magazine'.

```
result = collection.find(:journal =>
  'Oracle Magazine').update_one("$inc" => { :catalogId => 1 })
```

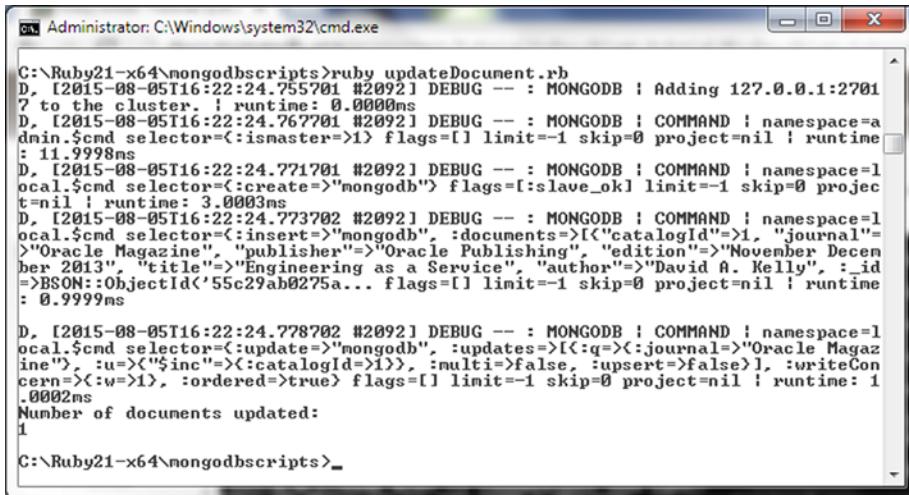
2. Output the number of documents updated.

```
print result.n
```

3. Run the `updateDocument.rb` script.

```
>ruby updateDocument.rb
```

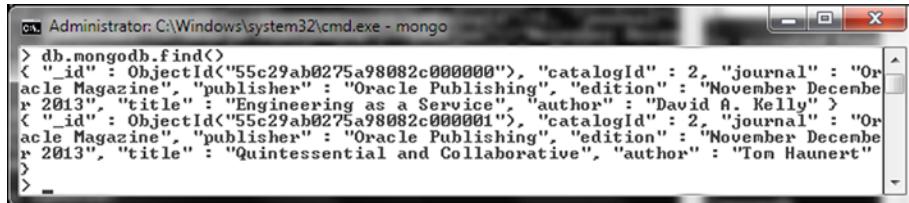
As the output shown in Figure 4-23 shows, one document gets updated.



```
C:\Ruby21-x64\mongodbscripts>ruby updateDocument.rb
D, [2015-08-05T16:22:24.755701 #20921] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. ! runtime: 0.0000ms
D, [2015-08-05T16:22:24.767701 #20921] DEBUG -- : MONGODB : COMMAND : namespace=a
dmn.$cmd selector=<:ismaster=1> flags=[] limit=-1 skip=0 project=nil ! runtime
: 11.9998ms
D, [2015-08-05T16:22:24.771701 #20921] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[{:slave_ok} limit=-1 skip=0 projec
t=nil ! runtime: 3.0003ms
D, [2015-08-05T16:22:24.773702 #20921] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":1, "journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decem
ber 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", :id
=> BSON::ObjectId('55c29ab0275a... flags=[] limit=-1 skip=0 project=nil ! runtime
: 0.9999ms
D, [2015-08-05T16:22:24.778702 #20921] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:update=>"mongodb", :updates=>[{:q=>{:journal:>"Oracle Magaz
ine"}, :u=>{"$inc":>{:catalogId=>1}}, :multi=>false, :upsert=>false}, :writeCon
cern=<:w=>1, :ordered=>true> flags=[] limit=-1 skip=0 project=nil ! runtime: 1
.0002ms
Number of documents updated:
1
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-23. Running the updateDocument.rb script

- Run the db.mongodb.find() command in Mongo shell to list the updated document as shown in Figure 4-24. Both documents have catalogId as 2; one document had catalogId 1 to start with and the other catalogId has been incremented by 1.



```
> db.mongodb.find()
{ "_id" : ObjectId("55c29ab0275a98082c000000"), "catalogId" : 2, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" }
{ "_id" : ObjectId("55c29ab0275a98082c000001"), "catalogId" : 2, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert"
}
>
```

Figure 4-24. One of the Document's catalogId Is incremented by 1 to 2

Example 2

- For the second example use the update_many method to update multiple documents. Use the \$set update operator to set publisher field to OraclePublishing. The find() method has a filter set to find all documents with publisher set to 'Oracle Publishing'.

```
result = collection.find(:publisher => 'Oracle Publishing').update_
many('$set' => { publisher: 'OraclePublishing'})
```

- Drop the `mongodb` collection with `db.mongodb.drop()`. Run the `updateDocument.rb` script. The output indicates that two documents have been updated as shown in Figure 4-25.

```
C:\Ruby21-x64\mongodbscripts>ruby updateDocument.rb
D, [2015-08-05T16:25:37.158706 #4948] DEBUG -- : MONGODB ! Adding 127.0.0.1:27017 to the cluster. : runtime: 0.0000ms
D, [2015-08-05T16:25:37.171706 #4948] DEBUG -- : MONGODB ! COMMAND : namespace=admindbs.cmd selector=<:ismaster=>1 flags=[1 limit=-1 skip=0 project=nil : runtime: 12.0010ns
D, [2015-08-05T16:25:37.174706 #4948] DEBUG -- : MONGODB ! COMMAND : namespace=local.$cmd selector=<:create=>"mongodb" flags=[{:slave_ok=1} limit=-1 skip=0 project=nil : runtime: 1.9999ms
D, [2015-08-05T16:25:37.176706 #4948] DEBUG -- : MONGODB ! COMMAND : namespace=local.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":1, "journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", :_id =>BSON::ObjectId<'55c29b71275a... flags=[1 limit=-1 skip=0 project=nil : runtime: 0.9999ms
D, [2015-08-05T16:25:37.178707 #4948] DEBUG -- : MONGODB ! COMMAND : namespace=local.$cmd selector=<:update=>"mongodb", :updates=>[{:q=>{:publisher=>"Oracle Publishing"}, :u=>{$set:">X:{:publisher=>"OraclePublishing"}, :multi=>true, :upsert=>false}], :writeConcern=>{:w=>1}, :ordered=>true} flags=[1 limit=-1 skip=0 project=nil : runtime: 1.0009ms
Number of documents updated:
2
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-25. Updating Multiple Documents

- Run the `db.mongodb.find()` method to list the two updated documents as shown in Figure 4-26.

```
> db.mongodb.find()
{ "_id" : ObjectId("55c29b71275a981354000000"), "catalogId" : 1, "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" }
{ "_id" : ObjectId("55c29b71275a981354000001"), "catalogId" : 2, "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" }
>
```

Figure 4-26. Listing Two Updated Documents

Example 3

- In the third example find the documents with edition set to November December 2013 and use the `replace_one()` method to replace one document with a document with only an edition field set to '11-12-2013'.

```
result = collection.find(:edition => 'November December 2013').
replace_one(:edition => '11-12-2013')
```

- Drop the `mongodb` collection with `db.mongodb.drop()`. Run the `updateDocument.rb` script with just Example 3. As the output indicates one document has been updated (replaced) as shown in Figure 4-27.

```
C:\Ruby21-x64\mongodbscripts>ruby updateDocument.rb
D, [2015-08-05T16:30:06.370103 #3152] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. | runtime: 0.0000ms
D, [2015-08-05T16:30:06.383104 #3152] DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<ismaster=1> flags=[] limit=-1 skip=0 project=nil | runtime
: 12.0010ms
D, [2015-08-05T16:30:06.387104 #3152] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil | runtime: 3.0000ms
D, [2015-08-05T16:30:06.389104 #3152] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":1, "journal"
:>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decem
ber 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", "_id
":> BSON::ObjectId<'55c29c7e275a... flags=[] limit=-1 skip=0 project=nil | runtime
: 1.0002ms
Number of documents updated:
1
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-27. Replacing One Document

- Run the db.mongodb.find() command in Mongo shell to list the replacement document as shown in Figure 4-28.

```
> db.mongodb.find()
{ "_id" : ObjectId("55c29c7e275a980c50000000"), "edition" : "11-12-2013" }
{ "_id" : ObjectId("55c29c7e275a980c50000001"), "catalogId" : 2, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert"
}
> -
```

Figure 4-28. Listing the Replaced Document

Example 4

- In the fourth example the find() method filter is set to find all documents with journal set to 'Oracle Magazine' and the find_one_and_replace() method is used to find and replace one document with a document with just the journal field set. The return_document is set to :after to return the document after replacement.

```
document = collection.find(:journal => 'Oracle Magazine').find_one_
and_replace({:journal => 'OracleMagazine'}, :return_document => :after)
```

Drop the mongodb collection with db.mongodb.drop(). When the updateDocument.rb script is run, one document gets replaced and the replaced document gets output as shown in Figure 4-29.

```
C:\Ruby21-x64\mongodbscripts>ruby updateDocument.rb
D, [2015-08-05T16:32:06.332965 #31921] DEBUG -- : MONGODB ! Adding 127.0.0.1:2701
7 to the cluster. i runtime: 0.0000ms
D, [2015-08-05T16:32:06.345966 #31921] DEBUG -- : MONGODB ! COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil i runtime
: 13.0010ns
D, [2015-08-05T16:32:06.350966 #31921] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil i runtime: 3.0000ns
D, [2015-08-05T16:32:06.351966 #31921] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":1, "journal"=
>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decem
ber 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", :_id
=>BSON::ObjectId<'55c29cf6275a980c78000000'>, flags=[] limit=-1 skip=0 project=nil i runtime
: 0.0000ms
D, [2015-08-05T16:32:06.353966 #31921] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:findandmodify=>"mongodb", :query=>{:journal=>"Oracle Magazi
ne"}, :update=>{:journal=>"OracleMagazine"}, :new=>true} flags=[:slave_ok] limit
=-1 skip=0 project=nil i runtime: 0.0000ns
Document after being updated:
{"_id":>BSON::ObjectId<'55c29cf6275a980c78000000'>, "journal":>"OracleMagazine"}
```

C:\Ruby21-x64\mongodbscripts>_

Figure 4-29. Finding and Replacing a Document

2. Drop the `mongodb` collection with `db.mongodb.drop()`. Run the `db.mongodb.find()` command in Mongo shell to list the documents including the replaced document as shown in Figure 4-30.

```
< "_id" : ObjectId("55c29cf6275a980c78000000"), "journal" : "OracleMagazine" >
< "_id" : ObjectId("55c29cf6275a980c78000001"), "catalogId" : 2, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert"
```

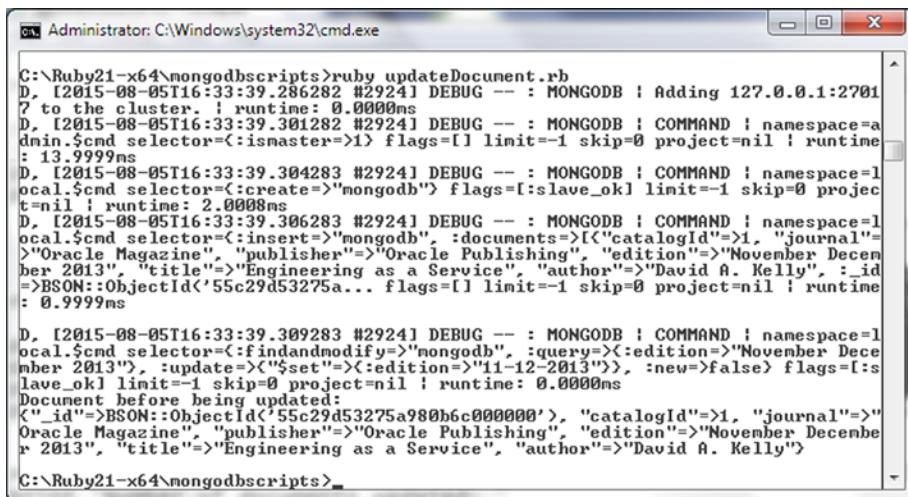
Figure 4-30. Listing the Replaced Document

Example 5

In the fifth example the `find()` method filter is set to find all documents with `edition` set to 'November December 2013' and the `find_one_and_update()` method is used to find and update the `edition` field using the `$set` operator.

```
document = collection.find(:edition => 'November December 2013').find_one_and_
update('$set' => {:edition=>'11-12-2013'})
```

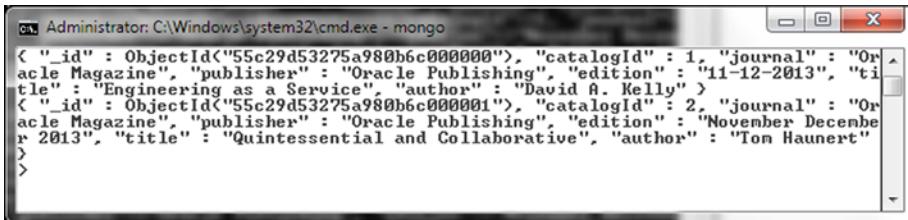
When the `updateDocument.rb` script is run, one of the documents with `edition` set to 'November December 2013' gets updated and the document before the update gets output, which is the default if `:return_document` is not set to `:after`, as shown in Figure 4-31.



```
C:\Ruby21-x64\mongodbscripts>ruby updateDocument.rb
D, [2015-08-05T16:33:39.286282 #2924] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. | runtime: 0.000ms
D, [2015-08-05T16:33:39.301282 #2924] DEBUG -- : MONGODB : COMMAND : namespace=admin.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil | runtime: 13.999ms
D, [2015-08-05T16:33:39.304283 #2924] DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:create=>"mongodb" flags=[{:slave_ok}] limit=-1 skip=0 project=nil | runtime: 2.000ms
D, [2015-08-05T16:33:39.306283 #2924] DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":1, "journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", :_id => BSON::ObjectId('55c29d53275a... flags=[] limit=-1 skip=0 project=nil | runtime: 0.999ms
D, [2015-08-05T16:33:39.309283 #2924] DEBUG -- : MONGODB : COMMAND : namespace=local.$cmd selector=<:findandmodify=>"mongodb", :query=<{:edition:>"November December 2013"}, :update=>{"$set":>{:edition:>"11-12-2013"}}, :new=>false> flags=[{:slave_ok}] limit=-1 skip=0 project=nil | runtime: 0.000ms
Document before being updated:
<"_id":>BSON::ObjectId('55c29d53275a980b6c000000'), "catalogId":1, "journal":>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November December 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly">
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-31. Finding and Updating a Document

When the db.mongodb.find() command is run in Mongo shell the updated document gets listed as shown in Figure 4-32.



```
C:\Administrator: C:\Windows\system32\cmd.exe - mongo
{
  "_id" : ObjectId("55c29d53275a980b6c000000"),
  "catalogId" : 1,
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "11-12-2013",
  "title" : "Engineering as a Service",
  "author" : "David A. Kelly"
},
{
  "_id" : ObjectId("55c29d53275a980b6c000001"),
  "catalogId" : 2,
  "journal" : "Oracle Magazine",
  "publisher" : "Oracle Publishing",
  "edition" : "November December 2013",
  "title" : "Quintessential and Collaborative",
  "author" : "Tom Haunert"
}
```

Figure 4-32. Listing the Updated Document

The updateDocument.rb script is listed with each of the five examples commented out. Run the script by uncommenting the example code to run.

```
require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

collection.create
```

```

document1={
  "catalogId" => 1,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}

document2={
  "catalogId" => 2,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}

collection.insert_many([document1,document2])

print "\n"

collection = client[:mongodb]

#Update example 1

#result = collection.find(:journal => 'Oracle Magazine').update_one("$inc" => { :catalogId => 1 })

#print "Number of documents updated: "
#print "\n"
#print result.n
#print "\n"

#Update example 2

#result = collection.find(:publisher => 'Oracle Publishing').update_many('$set' => { publisher: 'OraclePublishing' })
#print "Number of documents updated: "
#print "\n"
#print result.n
#print "\n"

#Update example 3

#result = collection.find(:edition => 'November December 2013').replace_one(:edition => '11-12-2013')
#print "Number of documents updated: "
#print "\n"
#print result.n
#print "\n"

```

```
#Update example 4
```

```
#document = collection.find(:journal => 'Oracle Magazine').find_one_and_replace
({:journal => 'OracleMagazine'}, :return_document => :after)
#print "Document after being updated: "
#print "\n"
#print document
#print "\n"
```

```
#Update example 5
```

```
#document = collection.find(:edition => 'November December 2013').find_one_and_update
('$set' => {:edition=>'11-12-2013'})
#print "Document before being updated: "
#print "\n"
#print document
#print "\n"
```

Deleting Documents

In this section we shall delete documents from a MongoDB collection. Methods for deleting document/s are included in Table 4-10 (see the “Finding Multiple Documents” section).

1. Create a Ruby script `deleteDocument.rb` and create a `Collection` instance for the `mongodb` collection.
2. Add four documents to the collection using the `insert_many()` method.

```
collection.create
document1={

    "catalogId" => 1,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Engineering as a Service",
    "author" => "David A. Kelly"
}

document2={

    "catalogId" => 2,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Quintessential and Collaborative",
    "author" => "Tom Haunert"
}
```

```

document3={

  "catalogId" => 3,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "",
  "author" => ""
}

document4={

  "catalogId" => 4,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "",
  "author" => ""
}

collection.insert_many([document1,document2,document3,document4])

```

3. Create a Collection instance over the `mongodb` collection.

```
collection = client[:mongodb]
```

4. Subsequently remove one of the documents using the `delete_one()` method. The `find()` method filter is set to find all documents with `edition` set to 'November December 2013'.

```
print collection.find(:edition => 'November December 2013').delete_one
```

As another example, delete a document using `find_one_and_delete()` method.

```
print collection.find(:journal => 'Oracle Magazine').find_one_and_delete
```

As a third example, delete multiple documents using the `delete_many()` method with the `find()` method filter set to find all documents with `edition` set to 'November December 2013'.

```
print collection.find(:edition => 'November December 2013').delete_many
```

5. The `deleteDocument.rb` script is listed below with some of the code commented out. First, uncomment the first two examples of deleting and run the script. Subsequently delete the `mongodb` collection and run the script with the third example to delete all documents.

```

require 'mongo'
include Mongo
client =Mongo::Client.new([ '127.0.0.1:27017' ], :database =>
'test')
client=client.use(:local)

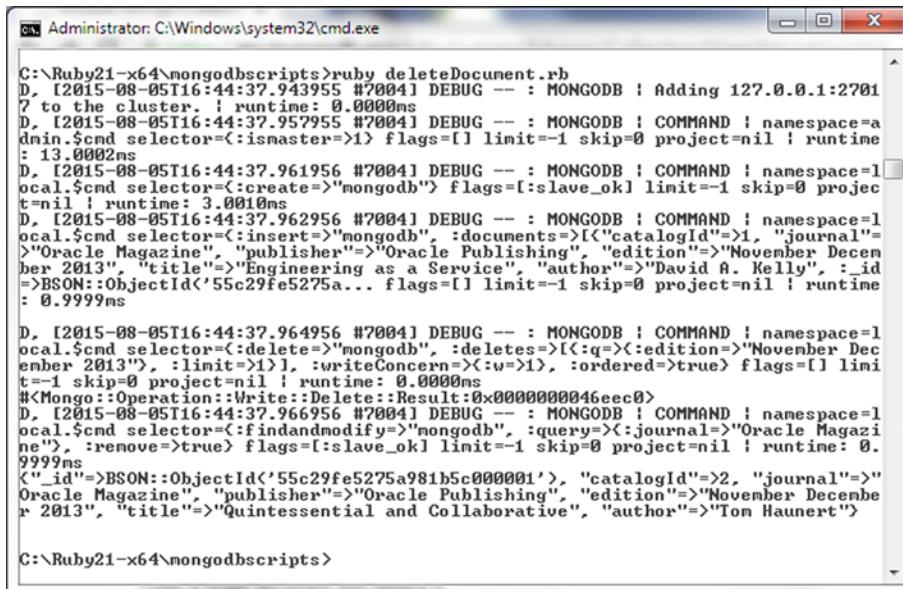
```

```

db=client.database
collection=db.collection("mongodb")
#collection.create
document1={
    "catalogId" => 1,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Engineering as a Service",
    "author" => "David A. Kelly"
}
document2={
    "catalogId" => 2,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "Quintessential and Collaborative",
    "author" => "Tom Haunert"
}
document3={
    "catalogId" => 3,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "",
    "author" => ""
}
document4={
    "catalogId" => 4,
    "journal" => "Oracle Magazine",
    "publisher" => "Oracle Publishing",
    "edition" => "November December 2013",
    "title" => "",
    "author" => ""
}
    collection.insert_many([document1,document2,document3,document4])
print "\n"
collection = client[:mongodb]
#print collection.find(:edition => 'November December 2013').delete_one
print "\n"
#print collection.find(:journal => 'Oracle Magazine').find_one_and_delete
print "\n"
#print collection.find(:edition => 'November December 2013').delete_many
print "\n"

```

6. Drop the `mongodb` collection with `db.mongodb.drop()`. Uncomment the `delete_one` and `find_one_and_delete` and run the `deleteDocument.rb` script to delete two of the four documents, one each with `delete_one` and `find_one_and_delete`. The output from the script is shown in Figure 4-33.



```
C:\Ruby21-x64\mongodbscripts>ruby deleteDocument.rb
D, [2015-08-05T16:44:37.943955 #7004] DEBUG -- : MONGODB ! Adding 127.0.0.1:2701
7 to the cluster. : runtime: 0.0000ms
D, [2015-08-05T16:44:37.957955 #7004] DEBUG -- : MONGODB ! COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1 flags=[] limit=-1 skip=0 project=nil : runtime
: 13.0002ns
D, [2015-08-05T16:44:37.961956 #7004] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:create=>"mongodb" flags=[:slave_ok] limit=-1 skip=0 projec
t=nil : runtime: 3.0010ms
D, [2015-08-05T16:44:37.962956 #7004] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb" :documents=>[{"catalogId":1, "journal"
:>"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decem
ber 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", :_id
=>BSON::ObjectId('55c29fe5275a... flags=[] limit=-1 skip=0 project=nil : runtime
: 0.9999ms
D, [2015-08-05T16:44:37.964956 #7004] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:delete=>"mongodb", :deletes=>[{:q=><:edition=>"November Dec
ember 2013"}, :limit=>1], :writeConcern=>{:w=>1}, :ordered=>true} flags=[] limi
t=-1 skip=0 project=nil : runtime: 0.0000ms
#<Mongo::Operation::Write::Delete::Result:0x0000000046ec0>
D, [2015-08-05T16:44:37.966956 #7004] DEBUG -- : MONGODB ! COMMAND : namespace=l
ocal.$cmd selector=<:findAndModify=>"mongodb", :query=><:journal=>"Oracle Magazi
ne", :remove=>true} flags=[:slave_ok] limit=-1 skip=0 project=nil : runtime: 0.
9999ms
{"_id":>BSON::ObjectId('55c29fe5275a981b5c000001'), "catalogId":>2, "journal":>"O
racle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decembe
r 2013", "title":>"Quintessential and Collaborative", "author":>"Tom Haunert"}>
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-33. Deleting Documents with `delete_one` and `find_one_and_delete`

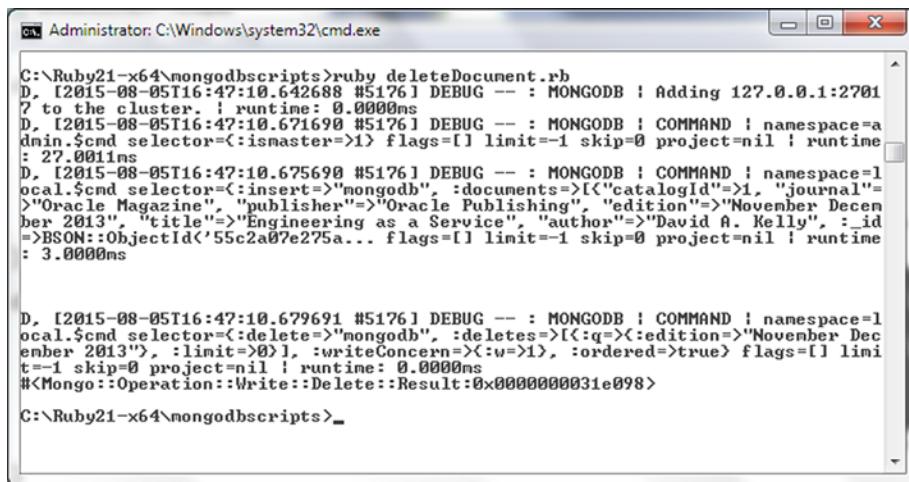
- Run the `db.mongodb.find()` command in Mongo shell to list two of the remaining documents as shown in Figure 4-34.



```
>
> db.mongodb.find()
{ "_id" : ObjectId("55c29fe5275a981b5c000002"), "catalogId" : 3, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "", "author" : "" }
{ "_id" : ObjectId("55c29fe5275a981b5c000003"), "catalogId" : 4, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "", "author" : "" }>
->
```

Figure 4-34. Listing Two of the Four Documents - Two Documents Deleted

- Delete the `mongodb` collection with the `db.mongodb.drop()` command in Mongo shell and subsequently run the `deleteDocument.rb` script again with just the third example of `delete_many`. The output from the `deleteDocument.rb` script is shown in Figure 4-35.



```
C:\Ruby21-x64\mongodbscripts>ruby deleteDocument.rb
D, [2015-08-05T16:47:10.642688 #5176] DEBUG -- : MONGODB : Adding 127.0.0.1:2701
? to the cluster. | runtime: 0.000ms
D, [2015-08-05T16:47:10.671690 #5176] DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=1> flags=[] limit=-1 skip=0 project=nil | runtime
: 27.001ms
D, [2015-08-05T16:47:10.675690 #5176] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":1, "journal":>
"Oracle Magazine", "publisher":>"Oracle Publishing", "edition":>"November Decem
ber 2013", "title":>"Engineering as a Service", "author":>"David A. Kelly", "_id
=>BSON::ObjectId('55c2a07e275a... flags=[] limit=-1 skip=0 project=nil | runtime
: 3.000ms

D, [2015-08-05T16:47:10.679691 #5176] DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:delete=>"mongodb", :deletes=>[{:q=>{}, :edition=>"November Dec
ember 2013"}, :limit=>0], :writeConcern=>{:w=>1}, :ordered=>true} flags=[] limi
t=-1 skip=0 project=nil | runtime: 0.0000ns
#<Mongo::Operation::Write::Delete::Result:0x0000000031e098>

C:\Ruby21-x64\mongodbscripts>
```

Figure 4-35. Deleting Multiple Documents with `delete_many`

When the `db.mongodb.find()` command is run in Mongo shell no document gets listed, as shown in Figure 4-36.



```
C:\Administrator: C:\Windows\system32\cmd.exe - mongo
> db.mongodb.find()
> -
```

Figure 4-36. Listing an Empty Collection

Performing Bulk Operations

The MongoDB Ruby driver supports bulk operations using the `bulk_write(operations, options)` method in the `Mongo::Collection` class. A list of operations may be invoked using the `bulk_write()` method. Each operation is defined with a document with one of the keys discussed in Table 4-11.

Table 4-11. Bulk Operations

Bulk Operation Key	Description
<code>insert_one</code>	Inserts one document.
<code>delete_one</code>	Deletes one document.
<code>delete_many</code>	Deletes many documents.
<code>replace_one</code>	Replaces one document.
<code>update_one</code>	Updates one document.
<code>update_many</code>	Updates many documents.

1. Create a Ruby script `bulk.rb` in the `C:\Ruby21-x64\mongodbscripts` directory.
2. Drop the `mongodb` collection if already present with `db.mongodb.drop()` and create the collection in the script.
3. Insert multiple documents using the `insert_many()` method.

```
collection.insert_many([document1,document2,document3,document4])
```

4. Create a collection for the `mongodb` collection.

```
collection = client[:mongodb]
```

5. Invoke the `bulk_write()` method with bulk operations `insert_one`, `update_one`, and `replace_one`. Set the second argument to `:ordered => true` to indicate that the bulk operations are to be performed in the specified order, which is also the default.

The `bulk.rb` script is listed below.

```
require 'mongo'
include Mongo

client =Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')
client=client.use(:local)
db=client.database
collection=db.collection("mongodb")

collection.create

document1={

  "catalogId" => 1,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Engineering as a Service",
  "author" => "David A. Kelly"
}

document2={

  "catalogId" => 2,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "Quintessential and Collaborative",
  "author" => "Tom Haunert"
}
```

```

document3={

  "catalogId" => 3,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "",
  "author" => ""

}

document4={

  "catalogId" => 4,
  "journal" => "Oracle Magazine",
  "publisher" => "Oracle Publishing",
  "edition" => "November December 2013",
  "title" => "",
  "author" => ""

}

collection.insert_many([document1,document2,document3,document4])

print "\n"

collection = client[:mongodb]

collection.bulk_write([
  { :insert_one => { :catalogId => 5 } },
  { :update_one => { :find => { :catalogId => 1 },
                     :update => { '$set' => { :catalogId => 6 } } }
  ],
  { :replace_one => { :find => { :catalogId => 2 },
                      :replacement => { :catalogId => 7 } }
  }
],
:ordered => true )

```

6. Run the `bulk.rb` script as follows.

```
>ruby bulk.rb
```

The output from the `bulk.rb` script is shown in Figure 4-37.

```
C:\Ruby21-x64\mongodbscripts>ruby bulk.rb
D. [2015-08-05T17:05:06.043198 #64521 DEBUG -- : MONGODB : Adding 127.0.0.1:27017 to the cluster. i runtime: 0.0000ns
D. [2015-08-05T17:05:06.061199 #64521 DEBUG -- : MONGODB : COMMAND : namespace=a
dmin.$cmd selector=<:ismaster=>1> flags=[] limit=-1 skip=0 project=nil i runtime
: 18.0013ms
D. [2015-08-05T17:05:06.067199 #64521 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":>1, "journal"=
>"Oracle Magazine", "publisher"=>"Oracle Publishing", "edition"=>"November Decem
ber 2013", "title"=>"Engineering as a Service", "author"=>"David A. Kelly", "_id"=>
BSON::ObjectId('55c2a4b2275a981934000004')}, flags=[] limit=-1 skip=0 project=nil i runtime
: 3.9999ms
D. [2015-08-05T17:05:06.068199 #64521 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:insert=>"mongodb", :documents=>[{"catalogId":>5, "_id"=>BSON:
:ObjectId('55c2a4b2275a981934000004')}, :writeConcern=<:w=>1>, :ordered=>true>
flags=[] limit=-1 skip=0 project=nil i runtime: 0.0000ms
D. [2015-08-05T17:05:06.070199 #64521 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:update=>"mongodb", :updates=>[{"q":>{"catalogId":>1}, "u":>{"$set":>{"catalogId":>6}}, :multi=>false, :upsert=>false}, :writeConcern=<:w=>1>,
:ordered=>true> flags=[] limit=-1 skip=0 project=nil i runtime: 0.0000ms
D. [2015-08-05T17:05:06.073200 #64521 DEBUG -- : MONGODB : COMMAND : namespace=l
ocal.$cmd selector=<:update=>"mongodb", :updates=>[{"q":>{"catalogId":>2}, "u":>{"c
atalogId":>7}, :multi=>false, :upsert=>false}, :writeConcern=<:w=>1>, :ordered=
>true> flags=[] limit=-1 skip=0 project=nil i runtime: 2.0013ms
C:\Ruby21-x64\mongodbscripts>
```

Figure 4-37. Running bulk.rb script

7. Subsequently list the documents in the `mongodb` collection using the `db.mongodb.find()` method. As shown in Figure 4-38, a new document with `catalogId` as 5 has been added. The `catalogId` 1 has been updated to 6 and the document with `catalogId` 2 has been replaced with a document with `catalogId` 7.

```
> db.mongodb.find()
{ "_id" : ObjectId("55c2a4b2275a981934000004"), "catalogId" : 6, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" }
{ "_id" : ObjectId("55c2a4b2275a981934000001"), "catalogId" : 7 }
{ "_id" : ObjectId("55c2a4b2275a981934000002"), "catalogId" : 3, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "", "author" : "" }
{ "_id" : ObjectId("55c2a4b2275a981934000003"), "catalogId" : 4, "journal" : "Or
acle Magazine", "publisher" : "Oracle Publishing", "edition" : "November Decembe
r 2013", "title" : "", "author" : "" }
{ "_id" : ObjectId("55c2a4b2275a981934000004"), "catalogId" : 5 }
```

Figure 4-38. Listing Documents after Bulk Operations

Summary

In this chapter we used the Ruby driver for MongoDB to connect to MongoDB Server and perform CRUD (create, read, update, and delete) operations on the database. We demonstrated each of the CRUD operations for both a single and multiple documents. In the next chapter we shall use the Node.js driver for MongoDB to connect to MongoDB and perform similar CRUD operations.

CHAPTER 5



Using MongoDB with Node.js

A traditional scripting language-based web application is built on the client/server model requiring a client scripting language such as JavaScript and a server scripting language such as PHP and making use of a web server. Node.js is different in that it is server side scripting built on V8 JavaScript Engine. V8 is Google's open source JavaScript engine used in Google Chrome. Node.js is based on the event-driven model and for developing fast, scalable, data-intensive, real-time, network applications. In this chapter we shall discuss accessing MongoDB with Node.js (or just Node) and making data modifications in the database using the Node.js driver for MongoDB. This chapter covers the following topics:

- Getting started
- Using a connection
- Using documents

Getting Started

In the following subsections we shall introduce the Node.js driver for MongoDB and set up the environment.

Overview of Node.js Driver for MongoDB

The Node.js driver for MongoDB provides an API to connect to MongoDB server and perform different operations in the server such as adding a document or finding a document. The main classes in the Node.js driver are illustrated in Figure 5-1.

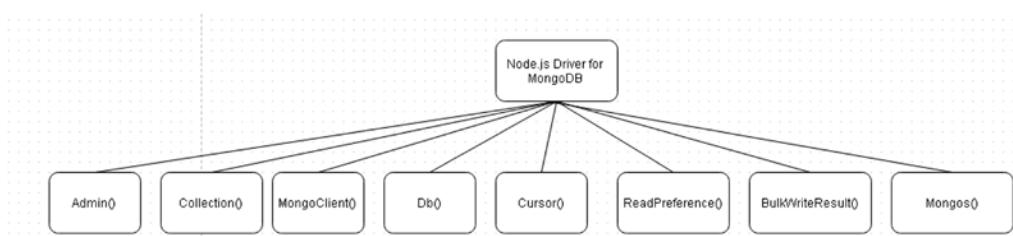


Figure 5-1. The Main Node.js Driver Classes

The Node.js driver classes are discussed in Table 5-1.

Table 5-1. The Main Node.js Driver Classes

Class	Description
Admin	Provides access to the admin functionality of MongoDB server such as retrieving the server info and status, adding/removing user, and authenticating.
Collection	Represents a collection in MongoDB server.
MongoClient	Represents a connection to MongoDB server.
Db	Represents a database instance.
Cursor	Represents a cursor over a query result set.
ReadPreference	Represents the read preference.
BulkWriteResult	Represents a bulk/batch write result.
Mongos	Represents an array of Mongo servers.
Server	Represents a MongoDB server.

Setting Up the Environment

We need to install the following software for this chapter:

- MongoDB
- Node.js
- Node.js driver for MongoDB

Installing MongoDB Server

Download MongoDB 3.0.5 from www.mongodb.org/ and extract the zip file to a directory. Add the bin directory from the MongoDB installation, for example, C:\Program Files\MongoDB\Server\3.0\bin to the PATH environment variable. Create the C:\data\db directory if not already created. Start MongoDB server with the following command.

```
>mongod
```

Installing Node.js

Download the node-v0.12.7-x64.msi application for Node.js from blog.nodejs.org/release/ and complete the following steps:

1. Double-click on the msi application to launch the Node.js Setup Wizard.
2. Click on Next in the Setup Wizard as shown in Figure 5-2.

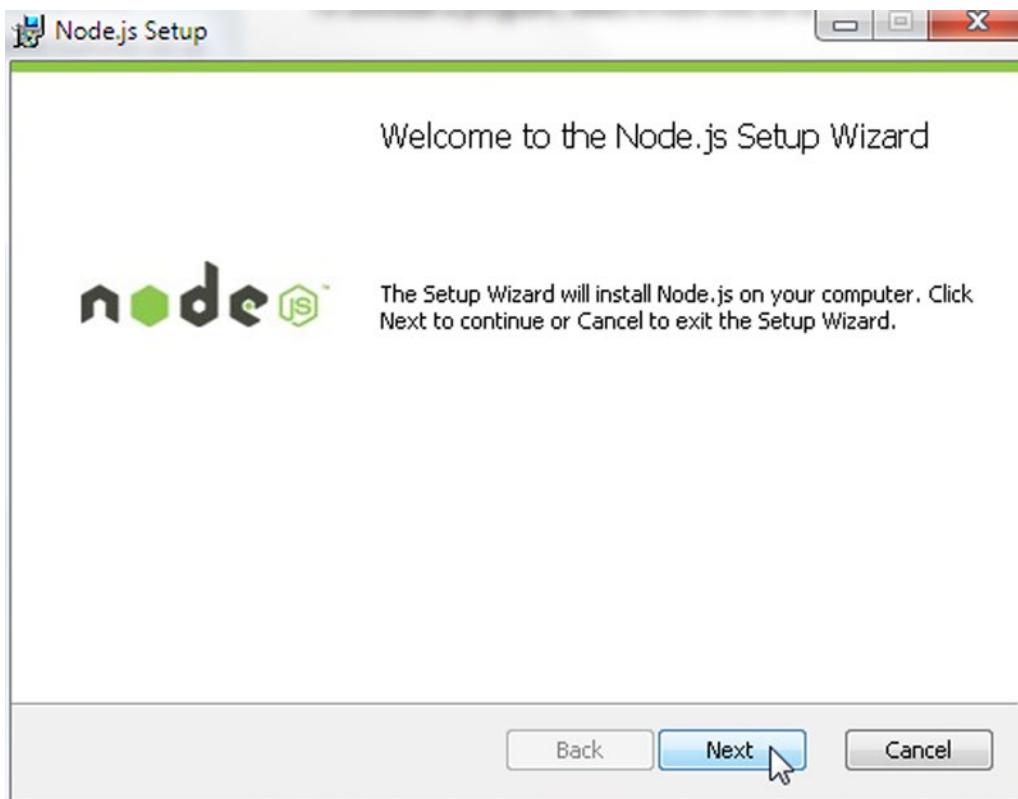


Figure 5-2. Node.js Setup Wizard

3. Accept the End-User License Agreement and click on Next.
4. In Destination Folder specify a directory to install Node.js in, the default being C:\Program Files\nodejs as shown in Figure 5-3. Click on Next.

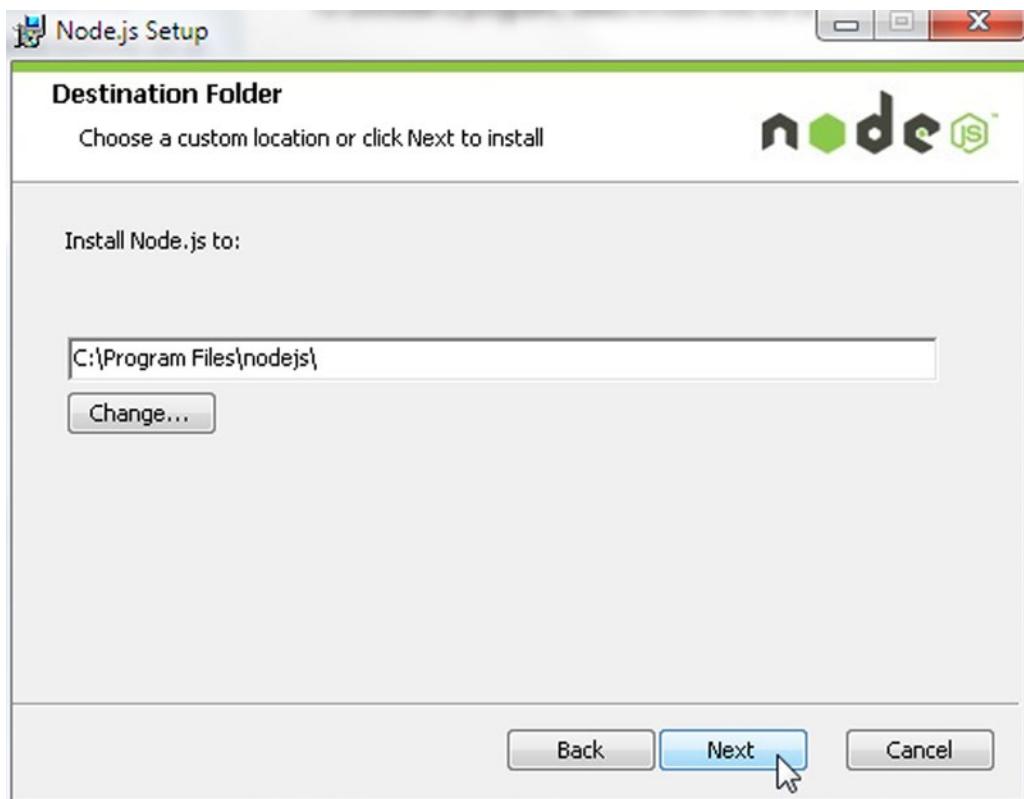


Figure 5-3. Selecting Installation Directory for Node.js

5. In Custom Setup, the Node.js features to be installed including the core Node.js runtime are listed for selection as shown in Figure 5-4. Choose the default settings and click on Next.

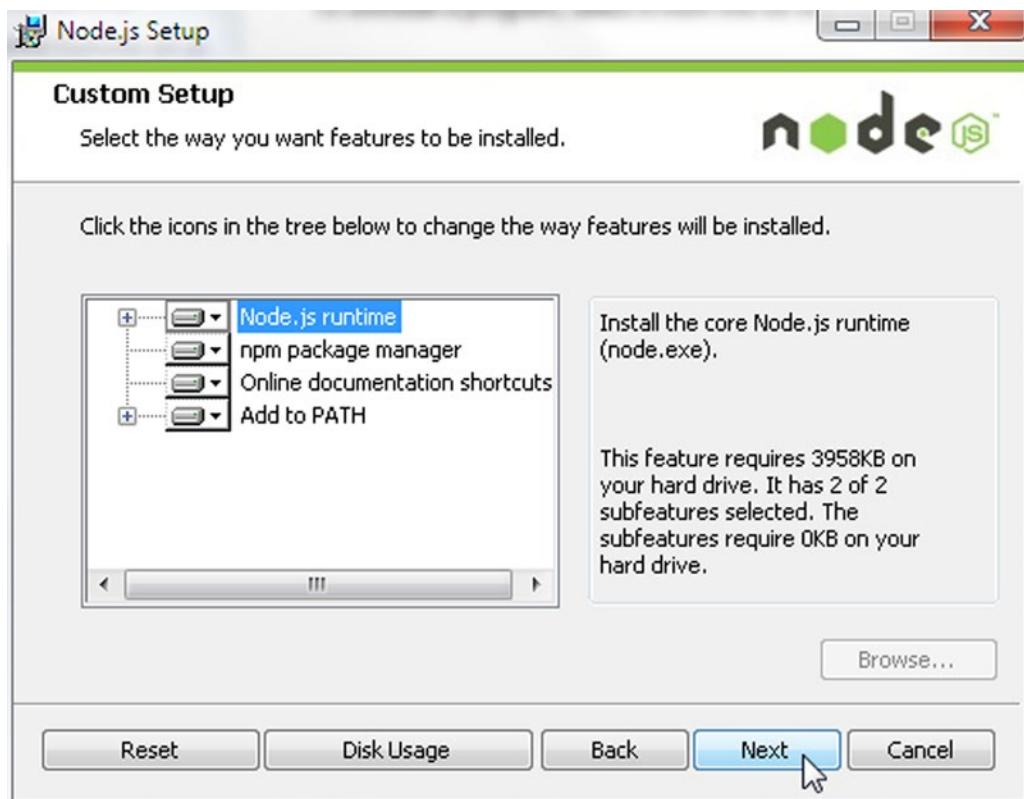


Figure 5-4. Selecting the Features to Install

6. In the Ready to Install Node.js window, click on Install as shown in Figure 5-5.

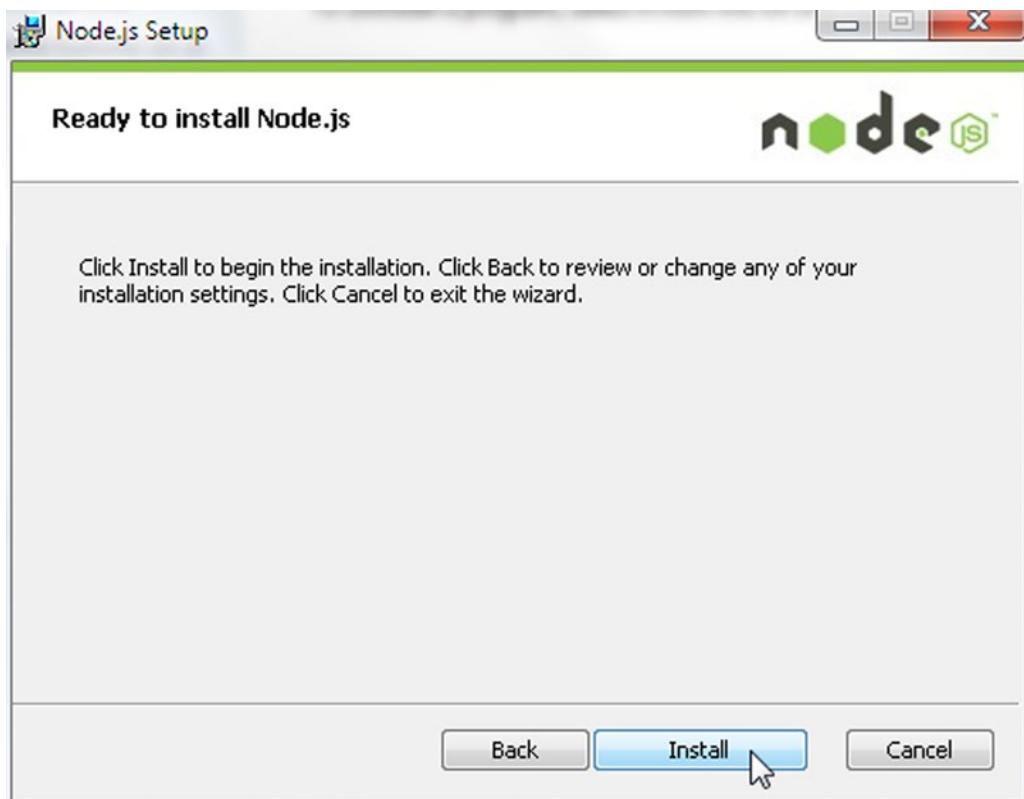


Figure 5-5. Clicking on *Install*

The installation of Node.js starts as shown in Figure 5-6. Wait for the installation to finish.

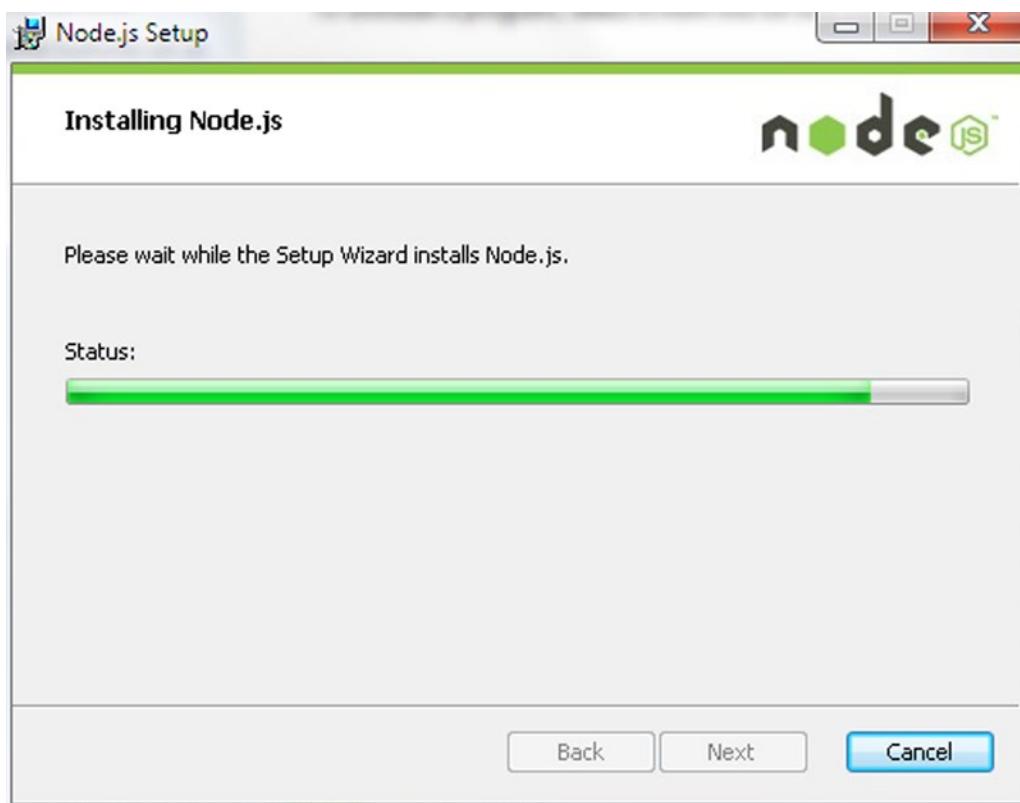


Figure 5-6. Installing Node.js

7. When the Node.js completes installing, click on Finish as shown in Figure 5-7.

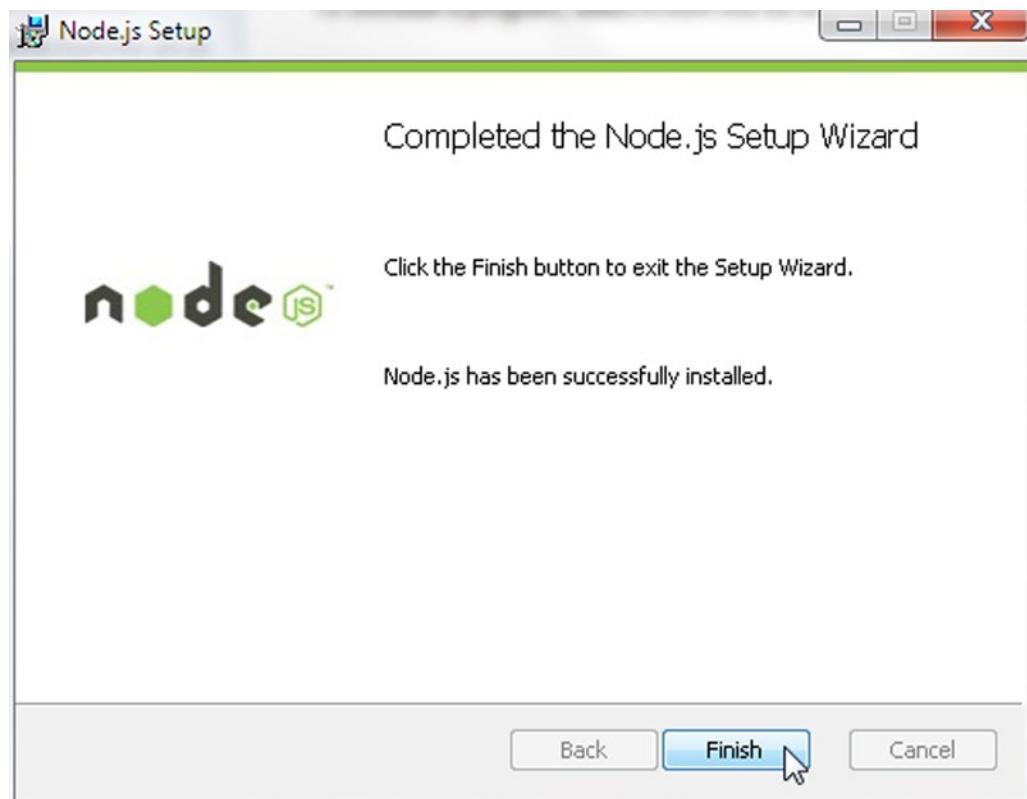


Figure 5-7. Node.js Installed

8. To find the version of Node.js installed, run the following command in a command shell.

```
node --version
```

The output from the command lists the version as 0.12.7 as shown in Figure 5-8.

A screenshot of a Windows Command Prompt window titled 'Administrator: C:\Windows\system32\cmd.exe'. The command 'node --version' is entered and the output 'v0.12.7' is displayed. The prompt then changes to 'C:\Users\Deepak Vohra>'. The window has standard Windows UI elements like minimize, maximize, and close buttons.

Figure 5-8. Finding the Node.js Version

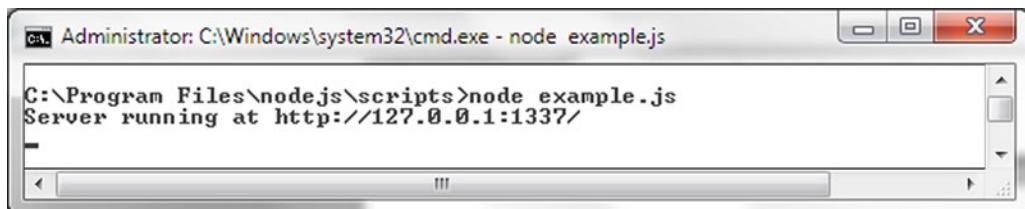
9. To test the Node.js installation, create a server using the following script; store the script in example.js in any directory, for example, the C:\Program Files\nodejs\scripts directory.

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

10. From the directory containing the script, run the script with the following command.

```
node example.js
```

The output from the script is shown in the command shell in Figure 5-9.



```
Administrator: C:\Windows\system32\cmd.exe - node example.js
C:\Program Files\nodejs\scripts>node example.js
Server running at http://127.0.0.1:1337/
```

Figure 5-9. Running the Node.js Example Script

Installing the Node.js Driver for MongoDB

Open a new terminal/console and go to C:\Program Files\nodejs. Then to install the Node.js driver for MongoDB run the following command.

```
npm install mongodb
```

Node.js driver for MongoDB gets installed as shown in Figure 5-10.

```
C:\Program Files\nodejs>npm install mongodb
> kerberos@0.0.12 install C:\Program Files\nodejs\node_modules\mongodb\node_modules\mongodb-core\node_modules\kerberos
> <node-gyp rebuild 2> builderror.log || (exit 0)

C:\Program Files\nodejs\node_modules\mongodb\node_modules\mongodb-core\node_modules\kerberos>if not defined npm_config_node_gyp <node "C:\Program Files\nodejs\node_modules\npm\bin\node-gyp-bin\\..\..\node_modules\node-gyp\bin\node-gyp.js" r
ebuild> else <node rebuild>
mongodb@2.0.40 node_modules\mongodb
└── readable-stream@1.0.31 <(isarray@0.0.1, inherits@2.0.1, string_decoder@0.10.3
1, core-util-is@1.0.1)
└── es6-promise@2.1.1
    └── mongodb-core@1.2.9 <(bson@0.4.11, kerberos@0.0.12)

C:\Program Files\nodejs>
```

Figure 5-10. Installing the Node.js Driver for MongoDB

Using a Connection

In the following subsections we shall create a connection with MongoDB server and create a database instance.

Creating a MongoDB Connection

In this section we shall connect to MongoDB server using the Node.js driver for MongoDB. We shall use the MongoClient class for connecting to MongoDB server. The MongoClient constructor does not take any args and has the following syntax.

```
MongoClient()
```

The MongoClient class supports the methods (static and instance) discussed in Table 5-2.

Table 5-2. MongoClient Class Methods

Method	Description
<code>MongoClient.connect(url, options, callback)</code>	Static method to connect to MongoDB server using a string URI. The connection URI would typically include the database name and has the following format. <code>mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]</code> The components of the connection URI are discussed in Table 5-3. The options parameter for the connect method is discussed in Table 5-6. The callback type is <code>connectCallback(error, db)</code>
<code>connect(url, options, callback)</code>	Instance method to connect to MongoDB server. The method parameters are the same as for the static method.

The components of the connection URI are discussed in Table 5-3.

Table 5-3. Components of the Connection URI

Component	Description
mongodb://	Required prefix in connection string.
username:password@	Login credentials for a specific database. Optional.
host1	MongoDB server address to connect to specified as a hostname, IP address, or UNIX domain socket. The only required component.
:port1	Port to connect on. Default is :27017.
host2[:port2],...[,hostN[:portN]]	Multiple host:port configurations for a replica set, for example.
/database	The database to authenticate connection to if username:password@ is specified. If username:password@ is specified but /database is not specified, connection to the admin database is authenticated.
?options	Connection string options. If /database is not specified a / must be added after the last hostN and ? Some of the connection string options are discussed in Table 5-4. Connection string options are specified as name/value pairs separated with an & or a, with the value being case sensitive.

The following are some of the simpler URI examples to connect to MongoDB server and are suitable for most purposes.

```

mongodb://
mongodb://localhost
mongodb://user1:password1@localhost
mongodb://user1:password1@localhost/test
mongodb://localhost:27017,localhost:27018,localhost:27019
mongodb://example1.com,example2.com

```

Some of the main connection string options are discussed in Table 5-4.

Table 5-4. Connection String Options

Option	Description
<code>uri.replicaSet</code>	Name of the replica set if the MongoDB is a member of the replica set. Used in conjunction with a seed list of at least two MongoDB instances.
<code>uri.connectTimeoutMS</code>	Connection timeout in milliseconds. Default is to never time out and implementation is driver specific.
<code>uri.maxPoolSize</code>	If the driver supports connection pooling and most drivers do, the maximum number of connections in the connection pool. Default is 100.
<code>uri.minPoolSize</code>	Minimum number of connections in the connection pool. Default is 0.
<code>uri.maxIdleTimeMS</code>	Maximum number of seconds a connection may stay idle in a connection pool before being closed. Option not supported by all drivers.
<code>uri.waitQueueMultiple</code>	Multiplied with <code>maxPoolSize</code> to provide the maximum number of threads that wait for a connection in a connection pool.
<code>uri.w</code>	Write concern option that specifies the level of write concern as a number or a string. The different levels of write concern are discussed in Table 5-5.
<code>uri.wtimeoutMS</code>	The wait time in milliseconds for the replication to succeed before timing out. Default is 0, which never times out.
<code>uri.journal</code>	If set to true write operations wait until MongoDB acknowledges the write operations and commits the data to the on disk journal. Default is false.
<code>uri.readPreference</code>	Applies to replica sets. Specifies the read preference mode. Value could be <code>primary</code> <code>primaryPreferred</code> <code>secondary</code> <code>secondaryPreferred</code> <code>nearest</code>
<code>uri.authSource</code>	Specifies the database name associated with the user credentials. Setting ignored if the connection string does not specify a user name and the value defaults to the database specified in the connection string.
<code>uri.authMechanism</code>	Authentication mechanism. Value could be <code>SCRAM-SHA-1</code> <code>MONGODB-CR</code> <code>MONGODB-X509</code> <code>GSSAPI</code> (Kerberos) <code>PLAIN</code> (LDAP SASL)

The write concern options describe the guarantees of a write operation such as whether the write operation has been committed. The different levels of the write concern are discussed in Table 5-5.

Table 5-5. Levels of Write Concern

Level	Description
0	The driver does not acknowledge write operations.
1	Basic acknowledgment level. A stand-alone MongoDB instance or the primary of a replica set acknowledges all write operations.
majority	Since MongoDB database version 3.0 the write operation returns only after the majority of the voting members of the replica set have acknowledged the write operation.
n	Applies to replica sets. The write operation returns only after the specified n number of servers in the replica set have acknowledged the write operation. Should not be set to a value greater than the replica set members as the write operation could wait indefinitely for the replica set members to become available.
tags	Applies to replica sets. Tag set configured with members of a replica set. The write operation waits until the replica set members with these tags configured have acknowledged the write operation.

The options parameter in the `connect(url, options, callback)` method supports the options discussed in Table 5-6.

Table 5-6. Options for Connect() Method

Name	Type	Description
<code>uri_decode_auth</code>	boolean	Specifies if the username and password are to be uri decoded. Default is <code>false</code> .
<code>db</code>	object	Hash of options to be set on the <code>Db</code> objects. Default value is <code>null</code> .
<code>server</code>	object	Hash of options to be set on the <code>server</code> objects. Default value is <code>null</code> .
<code>replSet</code>	object	Hash of options to be set on the <code>replSet</code> objects. Default value is <code>null</code> .
<code>mongos</code>	object	Hash of options to be set on the <code>mongos</code> objects. Default value is <code>null</code> .
<code>promiseLibrary</code>	object	An ES6-compatible Promise library for use by the application. Default value is <code>null</code> . A Promise is a wrapper function that may be used with Node functions that support a callback and a detailed discussion of promises. Promisification is beyond the scope of this chapter.

The `callback` parameter of the `connect()` method is of type `connectCallback(error, db)` and defines the callback format for results. The `error` arg is of type `MongoError` and represents an error instance. The `db` arg is of type `Db` and represents the connected database.

1. Create a `connection.js` script in the `C:\Program Files\nodejs\scripts` directory. The only method available to connect to MongoDB with Node is the `connect()` method, which has a static version and an instance version, and makes use of a connection URI. As listed before the syntax of the connection URI is as follows.

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

The required segment of the connection URL is `mongodb://host1`. The `port1` defaults to `:27017`. The `host2[:port2],...[,hostN[:portN]]` segment is not required if not using a replica set, which we won't be in this chapter. We won't also be using connection credentials `username:password`.

2. Import the `MongoClient` class from the `mongodb` module as follows using `require` statements.

```
MongoClient = require('mongodb').MongoClient;
```

3. Create a `MongoClient` instance.

```
var mongoclient = new MongoClient();
```

Next, we shall connect using the `connect(url, options, callback)` method in which the first parameter is the connection URI, the second the options, and the third the callback function. The callback function is called after the `connect()` method completes. The first arg of the callback function is an `MongoError` object if an error occurs or null and the second arg is an initialized `Db` object.

4. Connect with MongoDB server using the `connect()` method and in the connection URI specify the database as `test`. In the method block for the callback function log an error message to the console if an error occurs or log a message to indicate that the connection got established if an error does not occur.

```
mongoclient.connect("mongodb://localhost:27017/test", function
(error, db) {
    if (error)
        console.log(error);
    else
        console.log('Connected with MongoDB');
});
```

The `static` method `MongoClient.connect()` may also be used to connect to MongoDB server and supports the same parameters as the instance method `connect()`.

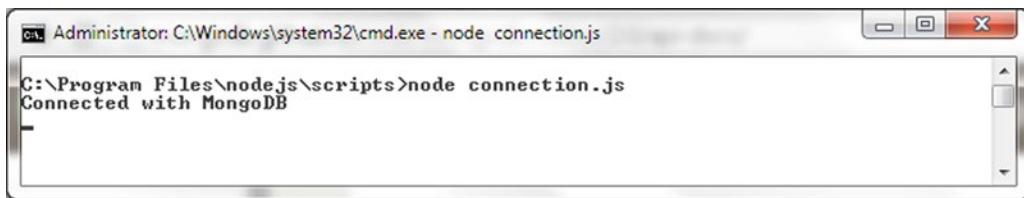
The `connection.js` script is listed:

```
MongoClient = require('mongodb').MongoClient;
var mongoclient = new MongoClient();
mongoclient.connect("mongodb://localhost:27017/test", function
(error, db) {
    if (error)
        console.log(error);
    else
        console.log('Connected with MongoDB');
});
```

5. Open a new terminal/console and navigate to `C:\Program Files\nodejs`. Run the `connection.js` script in Node.js with the following command.

```
>node connection.js
```

As the output in Figure 5-11 indicates, the method `connect()` establishes a connection with MongoDB.



```
Administrator: C:\Windows\system32\cmd.exe - node connection.js
C:\Program Files\nodejs\scripts>node connection.js
Connected with MongoDB
```

Figure 5-11. Connecting with MongoDB

Using the Database

In this section we shall discuss the `Db` class and create a MongoDB database instance. The `Db` class constructor has the following syntax.

`Db(databaseName, topology, options)`

The constructor parameters are discussed in Table 5-7.

Table 5-7. *Db* Constructor Parameters

Parameter	Type	Description
<code>databaseName</code>	string	The database name.
<code>topology</code>	Server ReplSet Mongos	The server topology for the server.
<code>options</code>	object	Optional settings.

The options supported by `Db` class constructor are discussed in Table 5-8.

Table 5-8. *Db Options*

Option	Type	Default	Description
authSource	string	null	Specifies another database name if the database authentication depends on another database.
w	number string	null	Write concern.
wtimeout	number		Write concern timeout.
j	boolean	false	Journal write concern.
native_parser	boolean	true	Specifies whether to use C++ bson parser instead of JavaScript parser.
forceServerObjectId	boolean	false	If set to true server assigns the _id values, not the driver.
serializeFunctions	boolean	false	Whether to serialize functions on any object.
raw	boolean	false	Specifies whether to return document results as raw bson buffers.
promoteLongs	boolean	true	Promote long values to number if they fit inside the 53-bit resolution.
bufferMaxEntries	number	-1	Sets an upper limit on the number of operations buffered by a driver while waiting for a connection. If the limit is exceeded the driver gives up on getting the connection. Default is -1, which is for unlimited.
numberOfRetries	number	5	Number of retries of connection.
retryMiliSeconds	number	500	Number of milliseconds between connections.
readPreference	ReadPreference string	null	The read preference. One of ReadPreference.PRIMARY, ReadPreference.PRIMARY_PREFERRED, ReadPreference.SECONDARY, ReadPreference.SECONDARY_PREFERRED, ReadPreference.NEAREST
pkFactory	object	null	A primary key factory object for generating custom _id keys.
promiseLibrary	object	null	A ES6-compatible Promise library.

The `Db` class provides the properties discussed in Table 5-9.

Table 5-9. *Db Class Properties*

Option	Type	Description
serverConfig	Server ReplSet Mongos	Gets the current topology.
bufferMaxEntries	number	Gets the current bufferMaxEntries value.
databaseName	string	The database name.
options	object	Options associated with the associated instance.
native_parser	boolean	Gets the current value of parameter native_parser.
slaveOk	boolean	Gets the current value of slaveOk.
writeConcern	object	Gets the current value of write concern.

Some of the methods supported by the Db class are discussed in Table 5-10.

Table 5-10. *Db Class Methods*

Method	Return Type	Description
addUser(username, password, options, callback)	Promise if no callback specified.	Adds a user to the database.
admin()	Admin db.	Returns the Admin instance.
authenticate(username, password, options, callback)	Promise if no callback specified.	Authenticates a user.
close(force, callback)	Object.	Closes the database and underlying connections.
collection(name, options, callback)	Collection.	Gets a collection.
collections(callback)	Promise if no callback specified.	Gets all collections for the current database instance.
command(command, options, callback)	Promise if no callback specified.	Runs a command.
createCollection(name, options, callback)	Promise if no callback specified.	Creates a collection.
db(name, options)	Db.	Creates a new database instance.
dropCollection(name, callback)	Promise if no callback specified.	Drops a collection.
dropDatabase(callback)	Promise if no callback specified.	Drops a database.
executeDbAdminCommand(command, options, callback)	Promise if no callback specified.	Runs a command as admin.
listCollections(filter, options)	CommandCursor.	Lists all collections.
logout(options, callback)	Promise if no callback specified.	Logs out user from server.
open(callback)	Promise if no callback specified.	Opens the database.
removeUser(username, options, callback)	Promise if no callback specified.	Removes a user from the database.
renameCollection(fromCollection, toCollection, options, callback)	Promise if no callback specified.	Renames a collection.

1. Create a JavaScript script db.js in the C:\Program Files\nodejs\scripts directory.
2. In the script we shall get a database instance and get a list of database collections from the database. First, import the MongoClient and Db classes from the mongodb module using require.

```
MongoClient = require('mongodb').MongoClient;
Db = require('mongodb').Db;
```

A Db instance is not required to be created directly using the class constructor but may be created implicitly by the MongoClient connect() method as the second parameter to callback function.

3. Create a MongoClient instance.

```
var mongoclient = new MongoClient();
```

4. Connect with the database using the connect(url, options, callback) method. Specify the URI as mongodb://localhost:27017/local, which includes the database instance as local. In the method block for the callback function, log an error message to the console if an error occurs; otherwise log the message 'Connected with MongoDB'.

```
mongoclient.connect("mongodb://localhost:27017/local",
function(error, db) {
    if (error)
        console.log(error);
    else
        console.log('Connected with MongoDB');
});
```

We shall use the listCollections(filter, options) method to list the collections in the local database. The listCollections(filter, options) may optionally specify a filter, for example, the filter {name: "collection1"} gets only the collection1 collection. The only option supported by listCollections() is batchSize. The listCollections() method returns a CommandCursor, which provides several methods including the toArray(callback) method to return an array of documents, the each(callback) method to iterate over the documents in the collection, and the next() method to get the next document.

5. Within the method block for the callback function of the connect() method, output the list of collections.

```
db.listCollections().toArray(function(err, items) {
    console.log(items);
    db.close();
});
```

The db.js script is listed below.

```
MongoClient = require('mongodb').MongoClient;
Db = require('mongodb').Db;
var mongoclient = new MongoClient();
mongoclient.connect("mongodb://localhost:27017/local", function(error, db) {
    if (error)
        console.log(error);
    else
        console.log('Connected with MongoDB');
    db.listCollections().toArray(function(err, items) {
        console.log(items);
        db.close();
    });
});
```

6. Open a new terminal/window and navigate to C:\Program Files\nodejs. Run the db.js script with the following command.

```
>node db.js
```

The different collections in the local database and their options get listed as shown in Figure 5-12.

```
C:\>Administrator: C:\Windows\system32\cmd.exe
C:\>Program Files\nodejs\scripts>node db.js
Connected with MongoDB
[ { name: 'mongodb', options: {} },
  { name: 'startup_log',
    options: { capped: true, size: 10485760 } },
  { name: 'system.indexes', options: {} } ]
C:\>Program Files\nodejs\scripts>_
```

Figure 5-12. Listing Collections in Local Database

The Db() class constructor creates a new database instance but does not create a new database as we shall demonstrate next.

1. Using the same script db.js, comment out the section of code for the mongoclient.connect() method invocation. Add a require statement for the Server class.

```
Server = require('mongodb').Server;
```

2. Create a new instance of Db using the class constructor new Db(databaseName, topology, options) with database name as 'mongo' and topology created using the Server class with host as localhost and port as 27017.

```
var db = new Db('mongo', new Server('localhost', 27017));
```

3. Open a connection with the database using the `open(callback)` method. The callback type for the `open()` method is `openCallback(error, db)`. Log error if generated. Output a list of collections using the `Db` instance returned if open is successful.

```
db.open(function(error, db) {
  if (error)
    console.log(error);
  db.listCollections().toArray(function(error, items) {
    console.log(items);
    db.close();
  });
});
```

The `db.js` used for the preceding example is listed below.

```
Db = require('mongodb').Db;
Server = require('mongodb').Server;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  db.listCollections().toArray(function(error, items) {
    console.log(items);
    db.close();
  });
});
```

4. Run the `db.js` script to list the collections in the `mongodb` database as shown in Figure 5-13.

```
C:\Program Files\nodejs\scripts>node db.js
[ { name: 'mongodb', options: {} },
  { name: 'startup_log',
    options: { capped: true, size: 10485760 } },
  { name: 'system.indexes', options: {} } ]
C:\Program Files\nodejs\scripts>
```

Figure 5-13. Running `db.js` Script with `Db` Instance Created Using Class Constructor

In the preceding example we used the `Server` topology. The topology could also be `Mongos` or `ReplSet`. For example, to use the `Mongos` topology, follow these steps:

1. Add a `require` statement for `Mongos` and create an instance of `Mongos` using an array of `Server` instances as follows. Not all the `Server` instances in the array have to be servers that are running and available. For example, we have listed a server at port 50000, but no MongoDB server at port 50000 is running when we run the example.

```
var mongos = new Mongos([
    new Server("localhost", 50000),
    new Server("localhost", 27017)
]);
```

2. Create an instance of Db using the Db class constructor with database name as the first argument, the Mongos class instance as the second argument.

```
var db = new Db('local', mongos);
```

3. Open the database using the open() method and in the callback function block list the collections as before.

The db.js script for the Mongos type topology is listed below.

```
Mongos = require('mongodb').Mongos;
Db = require('mongodb').Db;
Server = require('mongodb').Server;
var mongos = new Mongos([
    new Server("localhost", 50000),
    new Server("localhost", 27017)
]);
var db = new Db('local', mongos);
db.open(function(error, db) {
    if (error)
        console.log(error);
    db.listCollections().toArray(function(error, items) {
        console.log(items);
        db.close();
    });
});
```

4. Run the db.js script to list the collections in the mongodb database as shown in Figure 5-14.

```
C:\>Administrator: C:\Windows\system32\cmd.exe
C:\Program Files\nodejs\scripts>node db.js
[ { name: 'mongodbs', options: {} },
  { name: 'startup_log',
    options: { capped: true, size: 10485760 } },
  { name: 'system.indexes', options: {} } ]
C:\Program Files\nodejs\scripts>
```

Figure 5-14. Running db.js Script with Topology as Mongos

Using a Collection

A collection is represented with a `Collection` class object. The constructor for the `Collection` class does not take any args, and a new `Collection` instance may be created with `new Collection()`. The `Collection` class provides the properties discussed in Table 5-11.

Table 5-11. *Collection Class Properties*

Property	Type	Description
<code>collectionName</code>	string	Gets the collection name.
<code>namespace</code>	string	Gets the fully qualified collection namespace.
<code>writeConcern</code>	object	Gets the current write concern values.
<code>hint</code>	object	Gets the current index hint.

The `Collection` constructor is not meant to be invoked directly but is invoked internally. Some of the methods provided by the `Collection` class are discussed in Table 5-12.

Table 5-12. *Some Collection Class Methods*

Method	Return Type	Description
<code>bulkWrite(operations, options, callback)</code>	Promise if no callback specified.	Performs a bulk write operation.
<code>count(query, options, callback)</code>	Promise if no callback specified.	Counts the number of matching documents.
<code>createIndex(fieldOrSpec, options, callback)</code>	Promise if no callback specified.	Creates an index.
<code>deleteMany(filter, options, callback)</code>	Promise if no callback specified.	Deletes multiple documents.
<code>deleteOne(filter, options, callback)</code>	Promise if no callback specified.	Deletes a single document.
<code>distinct(key, query, options, callback)</code>	Promise if no callback specified.	Distinct documents for a given key.
<code>drop(callback)</code>	Promise if no callback specified.	Drops a collection.
<code>dropIndex(indexName, options, callback)</code>	Promise if no callback specified.	Drops an index.
<code>find(query)</code>	Cursor.	Finds documents. Creates a Cursor over the result set.
<code>findAndModify(query, sort, doc, options, callback)</code>	Promise if no callback specified.	Finds and updates a document. The method is deprecated and its alternative methods are <code>findOneAndUpdate</code> , <code>findOneAndReplace</code> or <code>findOneAndDelete</code> . The deprecated method could still be used.

(continued)

Table 5-12. (continued)

Method	Return Type	Description
<code>findAndRemove(query, sort, options, callback)</code>	Promise if no callback specified.	Finds and removes a document. The method is deprecated and its alternative method is <code>findOneAndDelete</code> . The deprecated method could still be used.
<code>findOne(query, options, callback)</code>	Promise if no callback specified.	Finds a single document.
<code>findOneAndDelete(filter, options, callback)</code>	Promise if no callback specified.	Finds a single document and deletes it.
<code>findOneAndReplace(filter, replacement, options, callback)</code>	Promise if no callback specified.	Finds a single document and replaces it.
<code>findOneAndUpdate(filter, update, options, callback)</code>	Promise if no callback specified.	Finds a single document and updates it.
<code>indexes(callback)</code>	Promise if no callback specified.	Returns all the indexes.
<code>insertMany(docs, options, callback)</code>	Promise if no callback specified.	Inserts an Array of Documents.
<code>insertOne(doc, options, callback)</code>	Promise if no callback specified.	Inserts a single document.
<code>isCapped(callback)</code>	Promise if no callback specified.	Finds if the collection is capped.
<code>listIndexes(options)</code>	CommandCursor.	Lists all the indexes.
<code>mapReduce(map, reduce, options, callback)</code>	Promise if no callback specified.	Runs a Map Reduce on the collection.
<code>options(callback)</code>	Promise if no callback specified.	Returns the options set of the collection.
<code>parallelCollectionScan(options, callback)</code>	Promise if no callback specified.	Returns n number of parallel cursors on the collection.
<code>rename(newName, options, callback)</code>	Promise if no callback specified.	Renames a collection.
<code>replaceOne(filter, doc, options, callback)</code>	Promise if no callback specified.	Replaces a Single Document.
<code>save(doc, options, callback)</code>	Promise if no callback specified.	Saves a document as a full document replacement. The method is deprecated and its alternative methods are <code>insertOne</code> , <code>insertMany</code> , <code>updateOne</code> or <code>updateMany</code> .
<code>stats(options, callback)</code>	Promise if no callback specified.	Gets all the collection stats.
<code>updateMany(filter, update, options, callback)</code>	Promise if no callback specified.	Updates multiple documents.
<code>updateOne(filter, update, options, callback)</code>	Promise if no callback specified.	Updates a single document.

The `Db` class provides several collection-related methods (`collection`, `collections`, `createCollection`, `dropCollection`, `listCollections`, `renameCollection`) as discussed in Table 5-10. In this section we shall use some of those methods to get, create, rename, and drop a collection and output collection-related information. We already used the `listCollections()` method in the preceding section.

1. Create a JavaScript script `collection.js` in the `C:\Program Files\nodejs\scripts` directory.
2. Import the required classes `MongoClient`, `Server`, and `Db`; some or all of these may be used in creating and accessing a collection based on how the `Db` instance is created.

```
MongoClient = require('mongodb').MongoClient;
Server = require('mongodb').Server;
Db = require('mongodb').Db;
```

3. Create a `Db` instance for the local database as discussed earlier.

```
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
    if (error)
        console.log(error);
});
```

4. Use the `db` instance in the callback function block for the `open()` method to invoke the `collections(callback)` method. In the callback function block of the `collections(callback)` method, output information about the collections.

```
db.collections(function(error, collections){
    if (error)
        console.log(error);
    else{
        console.log("Collections in database local");
        console.log(collections);
    }
});
```

5. Similarly, use the `db` instance in the callback function block for the `open()` method to invoke `collection(name, options, callback)` method to get the `mongodb` collection instance. In the callback function block of the `collection(name, options, callback)` method, output information about the collection such as the collection name, whether the collection capped, and the document count in the collection.

```
db.collection('mongodb', function(error, collection){
    if (error)
        console.log(error);
    else{
        console.log("Got collection from local database, collection name:
"+collection.collectionName);
```

```

collection.isCapped(function(error, result){
  console.log("Is collection capped?: " +result);
});
collection.count(function(error, result){
  console.log("Document count in the collection: "+result);
});
}
});

```

6. Next, create a collection called `mongo` using the `createCollection(name, options, callback)` method. Output the collection name using the callback function.

```

db.createCollection('mongo', function(error, collection){
  if (error)
    console.log(error);
  else{
    console.log("Collection created. Collection name: "+collection.
    collectionName); }
  });

```

7. Rename the `mongodb` collection to `mongocoll` using the `renameCollection(fromCollection, toCollection, options, callback)` method. Output the collection name returned in the callback function.

```

db.renameCollection('mongodb', 'mongocoll', function(error,
collection){
  if (error)
    console.log(error);
  else{
    console.log("Collection renamed: "+collection.collectionName);
  }
});

```

8. Drop the `mongocoll` collection using the `dropCollection(name, callback)` method.

```

db.dropCollection('mongocoll', function(error, result){
  if (error)
    console.log(error);
  else{
    console.log("Collection mongocoll dropped: "+result);
  }
});

```

The collection.js script is listed below.

```
MongoClient = require('mongodb').MongoClient;
Server = require('mongodb').Server;
Db = require('mongodb').Db;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  db.collections(function(error, collections){
    if (error)
      console.log(error);
    else{
      console.log("Collections in database local");
      console.log(collections);
    }
  });
  db.collection('mongodb', function(error, collection){
    if (error)
      console.log(error);
    else{
      console.log("Got collection from local database, collection name:
"+collection.collectionName);
      collection.isCapped(function(error, result){
        console.log("Is collection capped?: " +result);
      });
      collection.count(function(error, result){
        console.log("Document count in the collection: "+result);
      });
    }
  });
  db.createCollection('mongo', function(error, collection){
    if (error)
      console.log(error);
    else{
      console.log("Collection created. Collection name: "+collection.
collectionName); }
  });
  db.renameCollection('mongodb', 'mongocoll', function(error,
collection){
    if (error)
      console.log(error);
    else{
      console.log("Collection renamed: "+collection.collectionName);
    }
  });
});
```

```
db.dropCollection('mongocoll', function(error, result){  
    if (error)  
        console.log(error);  
    else{  
        console.log("Collection mongocoll dropped: "+result);  
    }  
});  
});
```

9. Before running the `collection.js` script create a collection called `mongodb` in the local database in the Mongo shell as shown in Figure 5-15. If the local database already has the `mongodb` collection the collection is not required to be created.

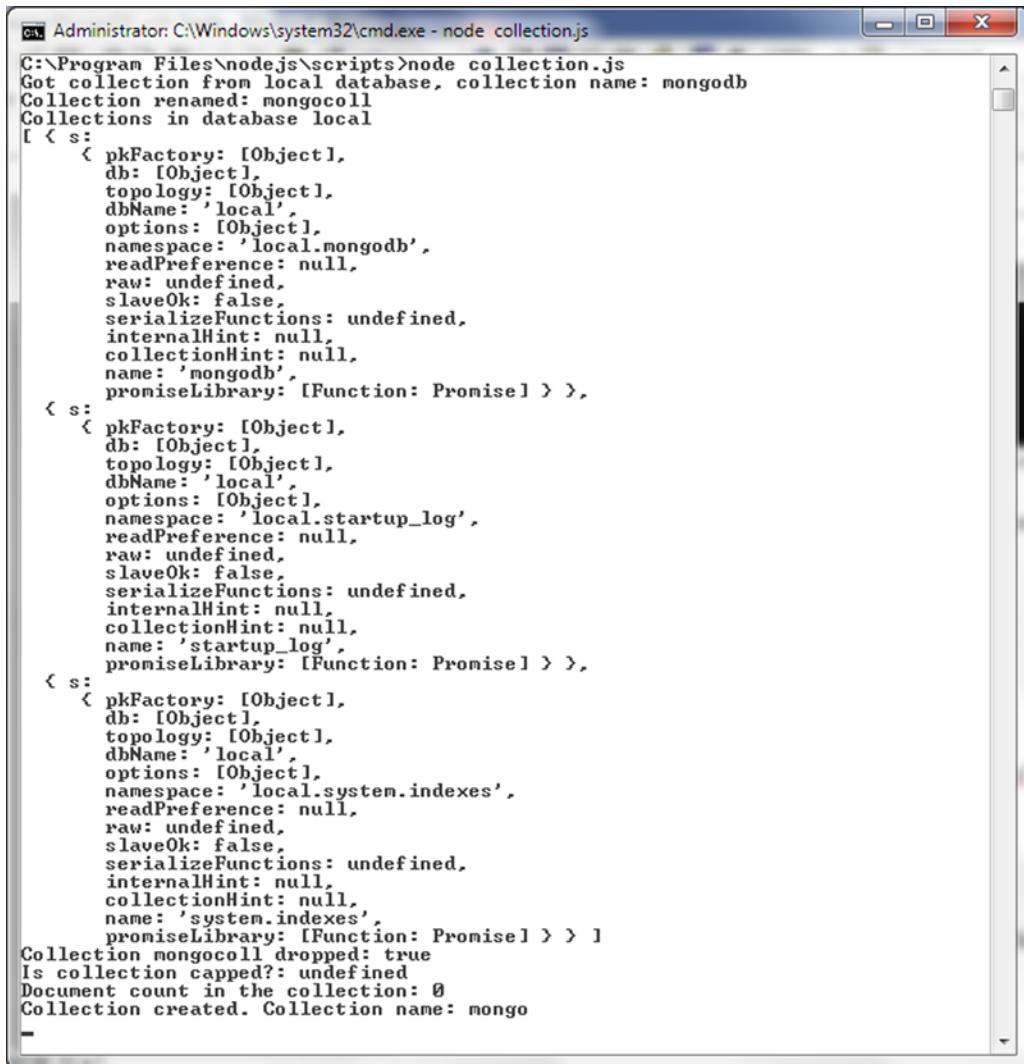
```
> show collections  
startup_log  
system.indexes  
> db.createCollection('mongodb')  
{ "ok" : 1 }  
> show collections  
mongodb  
startup_log  
system.indexes  
>
```

Figure 5-15. Creating the `mongodb` Collection

10. Run the `collection.js` script with the following command.

```
>node collection.js
```

The output from the script is shown in Figure 5-16. The script keeps running and to stop the script run `Ctrl + C`.



```

Administrator: C:\Windows\system32\cmd.exe - node collection.js
C:\Program Files\nodejs\scripts>node collection.js
Got collection from local database, collection name: mongodb
Collection renamed: mongocoll
Collections in database local
[ < s:
  { pkFactory: [Object],
    db: [Object],
    topology: [Object],
    dbName: 'local',
    options: [Object],
    namespace: 'local.mongodb',
    readPreference: null,
    raw: undefined,
    slaveOk: false,
    serializeFunctions: undefined,
    internalHint: null,
    collectionHint: null,
    name: 'mongodb',
    promiseLibrary: [Function: Promise] },
  < s:
  { pkFactory: [Object],
    db: [Object],
    topology: [Object],
    dbName: 'local',
    options: [Object],
    namespace: 'local.startup_log',
    readPreference: null,
    raw: undefined,
    slaveOk: false,
    serializeFunctions: undefined,
    internalHint: null,
    collectionHint: null,
    name: 'startup_log',
    promiseLibrary: [Function: Promise] },
  < s:
  { pkFactory: [Object],
    db: [Object],
    topology: [Object],
    dbName: 'local',
    options: [Object],
    namespace: 'local.system.indexes',
    readPreference: null,
    raw: undefined,
    slaveOk: false,
    serializeFunctions: undefined,
    internalHint: null,
    collectionHint: null,
    name: 'system.indexes',
    promiseLibrary: [Function: Promise] } ]
Collection mongocoll dropped: true
Is collection capped?: undefined
Document count in the collection: 0
Collection created. Collection name: mongo

```

Figure 5-16. Output from collection.js Script

We started off with the `mongodb` collection. Subsequently we created a collection called `mongo`. We renamed the `mongodb` collection to `mongocoll` and subsequently dropped the `mongocoll` collection. The only collection in the local database other than the `startup_log` and `system.indexes` collections is the `mongo` collection as shown in Figure 5-17.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
MongoDB shell version: 3.0.5
connecting to: test
> use local
switched to db local
> show collections
mongo
startup_log
system.indexes
> -
```

Figure 5-17. Listing Collections in Local Database after Running collection.js Script

Using Documents

In the following subsections we shall add a document, add a batch of documents, query documents, update documents, delete documents, and run bulk operations on documents.

Adding a Single Document

In this section we shall add a document to a MongoDB collection.

1. First, create a collection called `catalog` in the local database using the Mongo shell.

```
>db.createCollection('catalog')
```

The `catalog` collection gets created as listed with the `show collections` command in Mongo shell in Figure 5-18.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> show collections
mongo
startup_log
system.indexes
> db.createCollection('catalog')
{ "ok" : 1 }
> show collections
catalog
mongo
startup_log
system.indexes
> -
```

Figure 5-18. Creating the catalog Collection

2. Create an `addDocument.js` script in the `C:\Program Files\nodejs\scripts` directory. The `Collection` class provides the `insertOne(doc, options, callback)` method to add a single document.

3. In the addDocument.js script add require statements for the Db, Server, and Collection classes.

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
```

The parameters for the insertOne document are discussed in Table 5-13.

Table 5-13. Parameters for insertOne() Method

Parameter	Type	Description
doc	object	Document to insert.
options	object	Method options. The supported options are w (the write concern), wtimeout (the write concern timeout), j (Boolean indicating whether to enable journal write concern), serializeFunctions (Boolean indicating whether to serialize functions on any object), and forceServerObjectId(Boolean indicating whether the server assigns the _id values instead of the driver).
callback	insertWriteOpCallback (error, result)	The result callback function.

4. Create a Db instance using the class constructor.

```
var db = new Db('local', new Server('localhost', 27017));
```

5. Open the database using the open() method.

```
db.open(function(error, db) {
  //Get collection catalog
});
```

6. Invoke the collection() method of Db instance to get a catalog collection in the result callback function.

```
db.collection('catalog', function(error, collection){
  if (error)
    console.log(error);
  else{
    //Add document with insertOne
  }
});
```

7. Create JSON for a document to add.

```
doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December
2013',"title" : 'Engineering as a Service',"author" : 'David A. Kelly'};
```

8. Add the document using the `insertOne()` method.

```
collection.insertOne(doc1, function(error, result){
  if (error)
    console.log(error);
  else{
    console.log("Document added: "+result);
  }
});
```

The `addDocument.js` is listed.

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.collection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" :
'Oracle Publishing', "edition" : 'November December
2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
        collection.insertOne(doc1, function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Document added: "+result);
          }
        });
      });
    });
});
```

9. Run the `addDocument.js` script with the following command.

```
>node addDocument.js
```

A document gets added. The output from the script is shown in Figure 5-19.

The screenshot shows a Windows Command Prompt window titled 'Administrator: C:\Windows\system32\cmd.exe - node addDocument.js'. The command entered is 'node addDocument.js'. The output displayed is 'Document added: {"ok":1,"n":1}'.

Figure 5-19. Output from `addDocument.js`

- Run the following command in mongo shell to list the document added.

```
>use local
>db.catalog.find()
```

As shown in Figure 5-20 the added document gets listed.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
[{"_id": ObjectId("55c774f1de4cf8680b5600b1"), "catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}]
```

Figure 5-20. Listing Added Document

Adding Multiple Documents

In this section we add multiple documents to a MongoDB collection.

- Drop the catalog collection in the local database and create the catalog collection again as we shall be using the same collection to add multiple documents.

```
>use local
>db.catalog.drop()
>db.createCollection('catalog')
```

The Collection class provides the `insertMany(docs, options, callback)` method to add multiple documents. The method parameters are discussed in Table 5-14.

Table 5-14. Parameters for `insertMany` Method

Parameter	Type	Description
docs	Array <object>	Documents to insert.
options	object	Method options. The supported options are <code>w</code> (the write concern), <code>wtimeout</code> (the write concern timeout), <code>j</code> (Boolean indicating whether to enable journal write concern), <code>serializeFunctions</code> (Boolean indicating whether to serialize functions on any object), and <code>forceServerObjectId</code> (Boolean indicating whether the server assigns the <code>_id</code> values instead of the driver).
callback	<code>insertWriteOpCallback(error, result)</code>	The result callback function.

- Create a script `addDocuments.js` to add documents. Import the Server, Db, and Collection classes.

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
```

3. As in the preceding section on adding a single document obtain a Collection instance for the catalog collection. Create JSON for two documents to add.

```
doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013',
"title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
```

4. Using the insertMany() instance method from Collection class, add an array of documents.

```
collection.insertMany([doc1,doc2], function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

The addDocuments.js script is listed.

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
if (error)
    console.log(error);
else{
    db.collection('catalog', function(error, collection){

if (error)
    console.log(error);
else{
    doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" :
'Oracle Publishing', "edition" : 'November December
2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
    doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" :
'Oracle Publishing', "edition" : 'November December
2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
    collection.insertMany([doc1,doc2], function(error, result){
if (error)
    console.log(error);
else{
        console.log("Documents added: "+result);
    }
});}
});}
```

5. Drop and created the catalog collection and run the script with the following commands.

```
> db.catalog.drop()
>db.createCollection('catalog')
>node addDocuments.js
```

The output in Figure 5-21 shows that two documents get added.

```
Administrator: C:\Windows\system32\cmd.exe - node addDocuments.js
C:\Program Files\nodejs\scripts>node addDocuments.js
Documents added: [object Object]
```

Figure 5-21. Output from addDocuments.js

Run the db.catalog.find() command in Mongo shell to list the documents added as shown in Figure 5-22.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
>
> db.catalog.find()
[{"_id": ObjectId("55c778aa211d84001735384f"), "catalogId": "catalog1", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("55c778aa211d840017353850"), "catalogId": "catalog2", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}]
```

Figure 5-22. Listing Multiple Documents Added in Mongo Shell

Finding a Single Document

The Collection class provides the findOne(query, options, callback) method to find a single document from a collection. The method parameters are discussed in Table 5-15.

Table 5-15. Parameters for `findOne` Method

Parameter	Type	Description
query	object	Selector query to find a single document.
options	object	Method options.
callback	<code>resultCallback(error, result)</code>	The callback function. The first parameter of the callback function is the <code>MongoError</code> object if an error is generated and the second parameter is <code>result</code> .

Some of the options supported by the `findOne()` method are discussed in Table 5-16.

Table 5-16. Some of the Options for `findOne()` Method

Option	Type	Description	Default Value
limit	Number	Sets the limit of documents returned in the query.	0
sort	{Array Object}	Sorts the documents returned by the query.	null
fields	Object	Fields to include or exclude, not both. For example <code>{'a':1, 'b':1, 'c':1}</code> includes fields a, b, and c.	null
skip	Number	Skips the specified number of documents in the query. For example, if skip is 2 the query shall skip the first 2 documents that would otherwise be selected.	0
timeout	Boolean	Specifies if the cursor could timed out.	false
batchSize	Number	Sets the batch size for number of documents fetched in one network fetch when iterating over a query result set.	0
maxScan	Number	Limits the number of documents to scan.	null
showDiskLoc	Boolean	Whether to show the disk location of the results.	false
readPreference	ReadPreference	The preferred read preference.	null
maxTimeMS	Number	Number of milliseconds to wait before ending a query that is taking too long.	null

1. Create a script `findOneDocument.js` and import the required classes as before.
2. Create a `Db` instance and open the database instance.

3. Create a collection and add an array of documents to the collection. Within the method block for the callback function for the `createCollection()` method invoke the `findOne()` method. In the selector query specify the `journal` field set to 'Oracle Magazine'. In the options argument specify the fields to return as `edition`, `title`, and `author` and set the `skip` option to 1. In the callback function the second parameter is the cursor to the document returned by the `findOne()` method. Output the single document to the console.

```

collection.findOne({journal:'Oracle Magazine'}, {fields:{edition:1,title:1,author:1}, skip:1},
function(error, result) {
  if (error) console.log(error.message);
  else
    console.log(result);
  });
The findOneDocument.js script is listed:
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" : 'Engineering as a Service',"author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" : 'Quintessential and Collaborative',"author" : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
      collection.findOne({journal:'Oracle Magazine'}, {fields:{edition:1,title:1,author:1}, skip:1},
      function(error, result) {
        if (error) console.log(error.message);
        else
          console.log(result);
        });
      }
    });
  });
});
}
});
```

- Run the script with the command node `findOneDocument.js` in a command shell. A single document is output to the console as shown in Figure 5-23. Only the specified fields are output as the `fields` option was set.

```
C:\>Administrator: C:\Windows\system32\cmd.exe - node findOneDocument.js
C:\>Program Files\nodejs\scripts>node findOneDocument.js
Documents added: [Object Object]
{
  _id: 55c780dd4259b68c17f88ca0,
  edition: 'November December 2013',
  title: 'Quintessential and Collaborative',
  author: 'Tom Haunert' }
```

Figure 5-23. Finding One Document

Finding All Documents

In this section we shall find all documents from a collection.

- Drop the catalog collection from the local database as we shall be using the same collection in this section.

```
>use local
>db.catalog.drop()
```

While the `findOne` document returns a single document the `find(query)` method may be used to return one or more documents based on the selector query. The `find()` method has only one parameter, `query`, and returns a `Cursor`.

- Create a script `findAllDocuments.js` in the `C:\Program Files\nodejs\scripts` directory to find one or more or all documents. Import the required classes `Db`, `Collection`, and `Server`.
- Create and open a database instance and create a collection called `catalog` using the `createCollection()` method. Within the `createCollection()` method block add two documents to the collection using the `insertMany()` method. Invoke the `find()` method to find all documents. If a query is not specified all documents are selected by default. The `find()` method returns a `Cursor` over the result set. Invoke the `toArray(callback)` method on the `Cursor` object returned to return an array of documents. The callback function to the `toArray()` method has as the second parameter an array of BSON serialized objects.

```
collection.find().toArray(function(error, result) {
  if (error) console.warn(error.message);
  else
    console.log(result);
});
```

The `findAllDocuments.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.find().toArray(function(error, result) {
          if (error) console.warn(error.message);
          else
            console.log(result);
        });
      }
    });
});
```

4. Run the script with `node findAllDocuments.js` to find and list all documents as shown in Figure 5-24.

```
C:\>Administrator: C:\Windows\system32\cmd.exe - node findAllDocuments.js
C:\>Program Files\nodejs\scripts>node findAllDocuments.js
Documents added: [object Object]
[ { _id: 55c783e008c497c01cfbc4f4,
  catalogId: 'catalog1',
  journal: 'Oracle Magazine',
  publisher: 'Oracle Publishing',
  edition: 'November December 2013',
  title: 'Engineering as a Service',
  author: 'David A. Kelly' },
  { _id: 55c783e008c497c01cfbc4f5,
  catalogId: 'catalog2',
  journal: 'Oracle Magazine',
  publisher: 'Oracle Publishing',
  edition: 'November December 2013',
  title: 'Quintessential and Collaborative',
  author: 'Tom Haunert' } ]
```

Figure 5-24. Finding All Documents

Finding a Subset of Documents

In this section we shall find a subset of documents using the `find()` method already discussed earlier. We shall use the `skip`, `limit`, and `fields` options to skip some documents, limit the number of documents, and return a subset of fields.

1. Create a script `findSubsetDocuments.js`. Import the required classes `Collection`, `Server`, and `Db`.
2. Create and open a `Db` instance and create a collection called `catalog` as in earlier sections. Within the callback function block for the `createCollection()` method create an array of documents and add the documents to the collection using the `insertMany()` method.

```
doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December
2013',"title" : 'Engineering as a Service',"author" : 'David A.
Kelly'};
doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December
2013',"title" : 'Quintessential and Collaborative',"author" : 'Tom
Haunert'};
doc3 = {"catalogId" : 'catalog3', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December
2013'};
doc4 = {"catalogId" : 'catalog4', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December
2013'};
docArray=[doc1,doc2,doc3,doc4];
collection.insertMany(docArray, function(error, result){
    if (error)
        console.log(error);
    else{
        //Find subset of documents
    }
});
```

3. Within the callback function block for the `createCollection()` method invoke the `find()` method. The selector query argument is an empty document. Specify the `skip` option as 1 and the `limit` option as 2 and specify the `fields` option to include `edition`, `title`, and `author`. Use the `toArray()` method to send the query to the server. The second parameter of the callback function to the `toArray()` method is the document/s returned by the `find()` method query. Output the documents to the console.

```
collection.find({},{skip:1, limit:2,
fields:{edition:1,title:1,author:1}}).toArray(function(error,result) {
  if (error) console.log(error);
  else
    console.log(result);
});
```

The `findSubsetDocuments.js` is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013','title' : 'Engineering as a Service','author' : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013','title' : 'Quintessential and Collaborative','author' : 'Tom Haunert'};
        doc3 = {"catalogId" : 'catalog3', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
        doc4 = {"catalogId" : 'catalog4', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
        docArray=[doc1,doc2,doc3,doc4];
        collection.insertMany(docArray, function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.find({},{skip:1, limit:2, fields:{edition:1,title:1,author:1}}).toArray(function(error,result) {
```

```

if (error) console.log(error);
else
  console.log(result);
});
}
});
});

```

4. Drop the catalog collection from the local database.

```

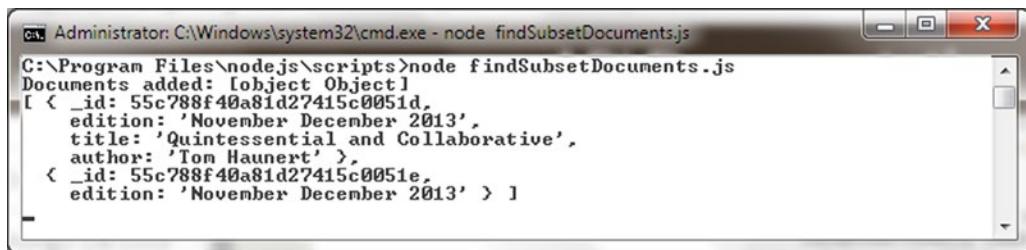
>use local
>db.catalog.drop()

```

Run the script with the following command.

```
>node findSubsetDocuments.js
```

Two different documents get returned and output to the console. Only the edition, title, and author fields get returned for the documents if the documents have those fields. One of the documents returned does not have the title and author fields, and only the edition field is returned as shown in Figure 5-25.



```

Administrator: C:\Windows\system32\cmd.exe - node findSubsetDocuments.js
C:\Program Files\nodejs\scripts>node findSubsetDocuments.js
Documents added: [Object Object]
[ { _id: 55c788f40a81d27415c0051d,
  edition: 'November December 2013',
  title: 'Quintessential and Collaborative',
  author: 'Tom Haunert' },
  { _id: 55c788f40a81d27415c0051e,
  edition: 'November December 2013' } ]

```

Figure 5-25. Finding a Subset of Documents

Using the Cursor

The `find()` method does not return the documents as such but returns a `Cursor` over the result set. The `Cursor` class provides several methods for getting more information about the documents such as iterating over the documents, counting the documents, and finding the next document in the result set. Some of the `Cursor` class methods are listed in Table 5-17.

Table 5-17. Some Cursor Class Methods

Method	Description
rewind()	Rewinds the cursor and returns the cursor. The cursor may again be iterated over to get the documents.
toArray(callback)	Returns an array of documents. If the cursor has been previously accessed the result contains only partial documents, the documents that have not yet been iterated over. The <code>rewind()</code> method must be called for a previously accessed cursor to get all the documents.
each(callback)	Iterates over the documents. If the cursor has been previously accessed not all documents are iterated over, only the documents that have not yet been iterated over are iterated. The <code>rewind()</code> method must be called for a previously accessed cursor to iterate over all the documents.
count(applySkipLimit, options, callback)	Returns the number of documents in the cursor. The Boolean <code>applySkipLimit</code> may be set to true to apply the <code>skip</code> and <code>limit</code> set on the cursor.
sort(keyOrList, direction)	Sorts the documents and returns a cursor over the sorted result set.
limit(value)	Limits the number of documents in the result set and returns a new cursor over the limited result set.
skip(value)	Skips the specified number of documents in the result set and returns a new cursor over the result set.
batchSize(value)	Sets the batch size and returns a new cursor.
nextObject(callback)	Gets the next document from the cursor.
close(callback)	Closes the cursor.

1. Create a script `findWithCursor.js` script in the `C:\Program Files\nodejs\scripts` directory.

2. Drop the `catalog` collection from the local database.

```
>use local
>db.catalog.drop()
```

3. Import the `Collection`, `Db`, and `Server` classes.

4. Create and open a `Db` instance for local database and subsequently create a `catalog` collection.

5. Within the callback function block for the `createCollection()` method block add some documents using the `insertMany()` method.

```
doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" :
'Engineering as a Service',"author" : 'David A. Kelly'};
doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" :
'Quintessential and Collaborative',"author" : 'Tom Haunert'};
docArray=[doc1,doc2];
collection.insertMany(docArray, function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

6. Invoke the `find()` method to obtain a `Cursor` object.

```
var cursor = collection.find();
```

7. Invoke the `each(callback)` method on the `Cursor` object to iterate over the documents. The second parameter in the callback function is the document being iterated over. Output the document to the console.

```
var cursor = collection.find();
cursor.each(function(error, result) {
    if (error) console.log(error);
    else
        console.log(result);
});
```

8. After the `each()` method has been invoked on the `Cursor` the `Cursor` cannot be used to iterate over the result set again. To demonstrate, invoke the `forEach()` method to iterate over the result set subsequent to invoking the `each()` method.

```
cursor.forEach(function(doc) {
    console.log(doc);
}, function(error) {
    console.log(error);
});
```

9. Next, we shall invoke another method on the Cursor to count the number of documents. Invoke the `count()` method and in the callback function the second parameter is the count of the documents. Output the count of the documents.

```
cursor.count(function(error, count) {
  if (error) console.log(error);
  else
    console.log("Document Count "+count);
});
```

The `findWithCursor.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
        docArray=[doc1,doc2];
        collection.insertMany(docArray, function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        var cursor = collection.find();
        cursor.each(function(error, result) {
          if (error) console.log(error);
          else
            console.log(result);
        });
        cursor.forEach(function(doc) {
          console.log(doc);
        }, function(error) {
          console.log(error);
        });
      }
    });
  }
});
```

```

cursor.count(function(error, count) {
  if (error) console.log(error);
  else
    console.log("Document Count "+count);
  });
});
});
});
```

10. Run the `findWithCursor.js` script with `node findWithCursor.js`. The output is shown in Figure 5-26.

```

Administrator: C:\Windows\system32\cmd.exe - node findWithCursor.js
C:\Program Files\nodejs\scripts>node findWithCursor.js
Documents added: [object Object]
{
  _id: '55c7979de633a1e81a4c1b72',
  catalogId: 'catalog1',
  journal: 'Oracle Magazine',
  publisher: 'Oracle Publishing',
  edition: 'November December 2013',
  title: 'Engineering as a Service',
  author: 'David A. Kelly' },
{
  _id: '55c7979de633a1e81a4c1b73',
  catalogId: 'catalog2',
  journal: 'Oracle Magazine',
  publisher: 'Oracle Publishing',
  edition: 'November December 2013',
  title: 'Quintessential and Collaborative',
  author: 'Tom Haunert' },
null
[Error: cursor is exhausted]
Document Count 2
```

Figure 5-26. Output from `findWithCursor.js` Script

When the `findWithCursor.js` script is run the two documents in the collection are returned when the `find()` method is invoked followed by the `each()` method. The `each()` method iterates over the Cursor to output the documents in the result set. When the `forEach()` method is invoked the cursor has already been exhausted and an error is output. The `count()` method invocation outputs the document count.

Finding and Modifying a Single Document

In this section we shall find and modify a document using the `findAndModify(query, sort, doc, options, callback)` method in `Collection` class. In this chapter we discuss both the deprecated method `findAndModify()` and its alternatives `findOneAndUpdate()`, `findOneAndReplace()`, or `findOneAndDelete()`. The `findAndModify()` method returns the original document before modification. The method parameters are discussed in Table 5-18.

Table 5-18. Parameters for *findAndModify()* Method

Parameter	Type	Description
query	object	Query to find the document to modify.
sort	array	If multiple documents match the query chooses the first one in the specified order.
doc	object	The document containing the fields and values to be updated.
options	object	Method options.
callback	resultCallback (error, result)	The callback function. The first parameter is the <code>MongoError</code> object if an error occurs or null and the second parameter is the result of the find and modify.

In addition to the `w`, `wtimeout`, and `j` options the following options listed in Table 5-19 are also supported.

Table 5-19. Options for *findAndModify()* Method

Parameter	Type	Description	Default Value
<code>remove</code>	boolean	Removes the object found and updated if remove set to true. If remove is also set and new is also set, new is ignored.	false
<code>upsert</code>	boolean	Adds a new document if a matching document is not found. The new document added is the replacement document in the third argument to the <code>findAndModify()</code> method.	false
<code>new</code>	boolean	Returns the new modified document. If remove is also set and new is also set, new is ignored.	false
<code>fields</code>	object	Fields projection for the result.	null

1. Create a script `findAndModify.js` in the `C:\Program Files\nodejs\scripts` directory.
2. Drop the `catalog` collection in the local database with `db.catalog.drop()`.
3. Add some documents to a collection as in earlier sections. Use numerical values for the `catalogId` and `edition` fields to demonstrate sorting.

```
doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : '11122013','title' :
'Engineering as a Service','author' : 'David A. Kelly'};
doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : '11122013','title' :
'Quintessential and
Collaborative','author' : 'Tom Haunert'};
collection.insertMany([doc1,doc2], function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

4. Within the callback function to the `createCollection()` method invoke the `findAndModify()` method. The query argument is the document `{journal:'Oracle Magazine'}`. The sort argument is the array `[['catalogId', 'ascending'], ['edition', 'descending']]`, which sorts by catalogId in ascending order and edition in descending order. In the update document set the edition and journal fields to new values using `{edition: '11-12-2013', journal:'OracleMagazine'}` as argument. In the options argument set the new option to true with `{new:true, w:1}`. In the callback function, log the updated document to the console.

```
collection.findAndModify({journal:'Oracle Magazine'},
  [['catalogId', 'ascending'], ['edition', 'descending']],
  {edition: '11-12-2013', journal:'OracleMagazine'}, {new:true,
  w:1}, function(error, result) {
    if (error) console.log(error);
  else
    console.log(result);
});
```

The `findAndModify.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine',
        "publisher" : 'Oracle Publishing', "edition" :
        '11122013',"title" : 'Engineering as a Service',"author" :
        'David A. Kelly'};
        doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine',
        "publisher" : 'Oracle Publishing', "edition" :
        '11122013',"title" : 'Quintessential and Collaborative',"author" :
        'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
      }
    });
  }
});
```

```

collection.findAndModify({journal:'Oracle Magazine'},
[['_catalogId', 'ascending'], ['edition', 'descending']],
{edition:'11-12-2013', journal:'OracleMagazine'}, {new:true,
w:1}, function(error, result) {
    if (error) console.log(error);
else
    console.log(result);
});
}
});
});

```

- Run the script with the command node findAndModify.js to find and modify a document and return the modified document as shown in Figure 5-27.

```

Administrator: C:\Windows\system32\cmd.exe - node findAndModify.js
Documents added: [object Object]
{
  value: {
    '_id': 55c79eeaaafae999c1689a383,
    'edition': '11-12-2013',
    'journal': 'OracleMagazine' },
  lastErrorObject: { updatedExisting: true, n: 1 },
  ok: 1
}

```

Figure 5-27. Output from findAndModify.js Script

- Run the db.catalog.find() method in Mongo shell to list the updated document. Only one document has been updated as shown in Figure 5-28. Because the catalogId field sort order is set to ascending, the first document found catalogId:1 is the document modified.

```

Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
> {
  "_id": ObjectId("55c7a028dde825c81aef6e7d"),
  "edition": "11-12-2013",
  "journal": "OracleMagazine"
}
{
  "_id": ObjectId("55c7a028dde825c81aef6e7e"),
  "catalogId": 2,
  "journal": "Oracle Magazine",
  "publisher": "Oracle Publishing",
  "edition": "11122013",
  "title": "Quintessential and Collaborative",
  "author": "Tom Haunert"
}
>

```

Figure 5-28. Listing the Updated Document

Finding and Removing a Single Document

The `findAndRemove(query, sort, options, callback)` method in Collection class is used to find and remove a document. In this chapter we have discussed both the deprecated method `findAndRemove()` and its alternative `findOneAndDelete()`. The method parameters are as follows in Table 5-20.

Table 5-20. Parameters for *findAndRemove()* Method

Parameter	Type	Description
query	object	Selector query to find the document to remove.
sort	array	If multiple documents match the query, the first in the specified order is selected for removal.
options	object	Method options. Supported options are w, wtimeout, and j.
callback	resultCallback (error, result)	The callback function. The first parameter is the Error object and the second parameter is the method result, which is the document removed.

1. Create a script `findAndRemove.js` in the `C:\Program Files\nodejs\scripts` directory.
2. Drop the `catalog` collection from the local database with `db.catalog.drop()`.
3. Add two documents to a collection as discussed before. The `catalogId` field should have a numerical value as we shall be sorting by the `catalogId` field.

```
doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 11122013,"title" : 'Engineering as a Service',"author" : 'David A. Kelly'};
doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 11122013,"title" : 'Quintessential and Collaborative',"author" : 'Tom Haunert'};
docArray=[doc1,doc2];
collection.insertMany(docArray, function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

4. Invoke the `findAndRemove()` method using the `{journal:'Oracle Magazine'}` as the query document and `[['catalogId', 'ascending']]` as the sort array. In the callback function the second parameter is the document to be removed. Log the document removed to the console.

```
collection.findAndRemove({journal:'Oracle Magazine'},
[['catalogId', 'ascending']], {w:1}, function(error, result) {
    if (error) console.log(error);
else
    console.log(result);
});
```

The `findAndRemove.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine',
        "publisher" : 'Oracle Publishing', "edition" : 11122013,"title"
        : 'Engineering as a Service','author' : 'David A. Kelly'};
        doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine',
        "publisher" : 'Oracle Publishing', "edition" : 11122013,"title"
        : 'Quintessential and Collaborative','author' : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.findAndRemove({journal:'Oracle Magazine'},
        [['catalogId', 'ascending']], {w:1}, function(error, result) {
          if (error) console.log(error);
        else
          console.log(result);
        });
      }
    });
});
```

- Run the `findAndRemove.js` script with the command `node findAndRemove.js`.

The document removed gets logged to the console as shown in Figure 5-29.

```
C:\Administrator: C:\Windows\system32\cmd.exe - node findAndRemove.js
Documents added: [object Object]
{ lastErrorObject: { n: 1 },
  value:
   { _id: 55c7a414929ab8b01ce35cb6,
     catalogId: 1,
     journal: 'Oracle Magazine',
     publisher: 'Oracle Publishing',
     edition: 11122013,
     title: 'Engineering as a Service',
     author: 'David A. Kelly' },
  ok: 1 }
```

Figure 5-29. Output from `findAndRemove.js` Script

Replacing a Single Document

The Collection class provides the `findOneAndReplace(filter, replacement, options, callback)` method to replace a single document. The method parameters are discussed in Table 5-21.

Table 5-21. Parameters for the `findOneAndReplace()` Method

Parameter	Type	Description
<code>filter</code>	object	Specifies the document selection filter to select the document/s to update.
<code>replacement</code>	object	The replacement document.
<code>options</code>	object	Method options.
<code>callback</code>	<code>resultCallback(error, result)</code>	The callback function, which has the first parameter a <code>MongoError</code> object and the second parameter the method result. The callback function must be specified if using write concern.

The options supported by the `findOneAndReplace()` method are discussed in Table 5-22.

Table 5-22. Options for the `findOneAndReplace()` Method

Option	Type	Description
<code>projection</code>	object	The fields. projection for the result.
<code>sort</code>	object	If the query selects multiple documents, determines which documents are replaced based on the sort order.
<code>maxTimeMS</code>	number	The maximum time in milliseconds for which the query may run.
<code>upsert</code>	boolean	Whether to upsert the document if it does not exist. Default is <code>false</code> .
<code>returnOriginal</code>	boolean	Whether to return the original document rather than the replacement document. Default is <code>true</code> .

1. Drop the catalog collection from the local database with `db.catalog.drop()`.
2. Create a `findOneAndReplaceDocument.js` script in the `C:\Program Files\nodejs\scripts` directory.
3. Get and open a `Db` instance, create a collection, and add documents to the collection.
4. Using the `findOneAndReplace(filter, replacement, options, callback)` method, replace one of the documents with journal field as Oracle Magazine. Provide a replacement document with `catalogId` as `catalog3`.

```
collection.findOneAndReplace({journal:'Oracle Magazine'}, {"catalogId" : 'catalog3', "journal" : 'OracleMagazine', "publisher" : 'OraclePublishing', "edition" : '11122013'}, function(error, result) {
  if (error) console.warn(error.message);
  else
    console.log(result);
});
```

The `findOneAndReplaceDocument.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.findOneAndReplace({journal:'Oracle Magazine'}, {"catalogId" : 'catalog3', "journal" : 'OracleMagazine', "publisher" : 'OraclePublishing', "edition" : '11122013'}, function(error, result) {
          if (error) console.warn(error.message);
          else
            console.log(result);
        });
      }
    });
});
```

5. Run the script to replace a single document with node `findOneAndReplaceDocument.js`. The original document that gets replaced gets returned and output as shown in Figure 5-30.

```
C:\ Administrator: C:\Windows\system32\cmd.exe - node findOneAndReplaceDocument.js
C:\Program Files\nodejs\scripts>node findOneAndReplaceDocument.js
Documents added: [object Object]
{
  value: {
    _id: '55c7a9b992a4bc5c0201592a',
    catalogId: 'catalog1',
    journal: 'Oracle Magazine',
    publisher: 'Oracle Publishing',
    edition: 'November December 2013',
    title: 'Engineering as a Service',
    author: 'David A. Kelly',
    lastErrorObject: { updatedExisting: true, n: 1 },
    ok: 1
  }
}
```

Figure 5-30. Output from *findOneAndReplaceDocument.js* Script

- Run a query in the Mongo shell with `db.catalog.find()` to list the documents including the replacement document as shown in Figure 5-31.

```
C:\ Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
{ "_id" : ObjectId("55c7aa4bdea63f7c1b6f36ea"), "catalogId" : "catalog3", "journal" : "OracleMagazine", "publisher" : "OraclePublishing", "edition" : "11122013" }
{ "_id" : ObjectId("55c7aa4bdea63f7c1b6f36eb"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" }
> -
```

Figure 5-31. Listing the Replacement Document

Another method that could be used to replace a document is the `replaceOne(filter, doc, options, callback)` method, which is similar to the `findOneAndReplace(filter, replacement, options, callback)` method except that the `replaceOne` method provides fewer options (`w`, `wtimeout`, `j`, `upsert`, and `upsert`).

Updating a Single Document

In this section we shall update a document. The `findOneAndUpdate(filter, update, options, callback)` method is used to update one or more documents based on a selector query. The method has the parameters shown in Table 5-23.

Table 5-23. Parameters for the *findOneAndUpdate()* Method

Parameter	Type	Description
filter	object	Specifies the document selection filter to select the document/s to update.
update	object	Update operations to be performed on the document.
options	object	Method options.
callback	resultCallback (error, result)	The callback function, which has the first parameter a <code>MongoError</code> object and the second parameter the method result. The callback function must be specified if using write concern.

The options supported by the `findOneAndUpdate()` method are the same as for the `findOneAndReplace()` method and discussed in the preceding section in Table 5-21.

1. Drop the `catalog` collection in the local database with `db.catalog.drop()`.
2. Create an `updateDocument.js` script in the `C:\Program Files\nodejs\scripts` directory.
3. Create a collection called `catalog` as discussed previously.
4. Add two documents with only some of the fields set as we shall be adding others as an update.

```
doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine',
"publisher" : 'Oracle Publishing', "edition" : 'November December 2013'};
collection.insertMany([doc1,doc2], function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

5. Using the `findOneAndUpdate(filter, update, options, callback)` method with filter as all documents with journal field as 'Java Magazine' update one of the documents set to `{journal:'Java Magazine'}` using the `$set` operator. Set the `upsert` option to true so that a new document is added if one not found.

```
collection.findOneAndUpdate({journal:'Java Magazine'}
, {$set: {journal:'Java Magazine'}}
, {returnOriginal: false
, upsert: true
}
, function(error, result) {
if (error) console.log(error);
else
    console.log(result);
});
```

The second argument for the fields and values to be updated may be specified using any of the update operators. When using the update operator `$set` to modify field values, not all the fields/values need to be specified. Only the fields to be updated need to be specified.

6. As another example, update the first matching document that has the journal field set to 'Oracle Magazine', using the \$set operator in the document argument to update the edition, title, and author fields. As the documents added with insertMany do not include the title and author fields these fields are added when the update is made. In the callback function, output the method result.

```
collection.update({journal:'Oracle Magazine'},
{$set:{edition:'11-12-2013', title:'Engineering as a Service',
author:'David A. Kelly'}}, {upsert:true, w: 1},
function(err, result){
if (err)
    console.log(err);
else
    console.log(result);
});
```

The updateDocument.js script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
if (error)
    console.log(error);
else{
db.createCollection('catalog', function(error, collection){
if (error)
    console.log(error);
else{
doc1 = {"catalogId" : 1, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing',
"edition" : 'November December 2013'};
doc2 = {"catalogId" : 2, "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing',
"edition" : 'November December 2013'};
collection.insertMany([doc1,doc2], function(error, result){
if (error)
    console.log(error);
else{
console.log("Documents added: "+result);
}
});
collection.findOneAndUpdate({journal:'Java Magazine'}
, {$set: {journal:'Java Magazine'}}
, {returnOriginal: false
,upsert: true
}
, function(error, result) {
if (error) console.log(error);
else
    console.log(result);
});
```

```

collection.update({journal:'Oracle Magazine'}, {$set:{edition:'11-12-2013',
title:'Engineering as a Service', author:'David A.
Kelly'}}, {upsert:true},
function(error, result) {
  if (error) console.log(error);
  else
    console.log(result);

  });
}
});
});
});

```

- Run the script with the command node updateDocument.js. The output shown in Figure 5-32 indicates that one document gets upserted.

```

Administrator: C:\Windows\system32\cmd.exe - node updateDocument.js
C:\Program Files\nodejs\scripts>node updateDocument.js
Documents added: 1
{
  value: {
    _id: 55c7b825aaaf73f3a8d7afb85,
    journal: 'Java Magazine',
    catalogId: 1,
    edition: '11-12-2013',
    title: 'Engineering as a Service',
    author: 'David A. Kelly',
    publisher: 'Oracle Publishing'
  },
  lastErrorObject: {
    updatedExisting: false,
    n: 1,
    upserted: 55c7b825aaaf73f3a8d7afb85
  },
  ok: 1
}
{
  result: {
    ok: 1,
    nModified: 1,
    n: 1
  }
}

```

Figure 5-32. Output from updateDocument.js

- Run the db.catalog.find() method in Mongo shell to list the document in the collection. One of the documents listed is the upserted document as shown in Figure 5-33. Another document has been updated with some fields added.

```

Administrator: C:\Windows\system32\cmd.exe - mongo
>
> db.catalog.find()
[{"_id": ObjectId("55c7b825aaaf73f3a8d7afb85"), "catalogId": 1, "journal": "Java Magazine", "publisher": "Oracle Publishing", "edition": "11-12-2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("55c7b825079d13941c372d22"), "catalogId": 2, "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November December 2013"}, {"_id": ObjectId("55c7b825aaaf73f3a8d7afb85"), "catalogId": 1, "journal": "Java Magazine"}]
>

```

Figure 5-33. Listing Updated Documents

In the updated document (not the upserted document) only the fields specified in the \$set operator are updated and the other fields are the same as before the update. Only the first matching document is updated.

Updating Multiple Documents

In this document we shall update multiple documents using the `updateMany(filter, update, options, callback)` method. The parameters are discussed in Table 5-24.

Table 5-24. Parameters for the `updateMany()` Method

Parameter	Type	Description
<code>filter</code>	object	Specifies the document selection filter to select the document/s to update.
<code>update</code>	object	Update operations to be performed on the document.
<code>options</code>	object	Method options. The options are <code>upsert</code> , <code>w</code> , <code>wtimeout</code> , and <code>j</code> .
<code>callback</code>	<code>writeOpCallback(error, result)</code>	The callback function, which has the first parameter a <code>MongoError</code> object and the second parameter the method result. The callback function must be specified if using write concern.

1. Create a script `updateDocuments.js` in the `C:\Program Files\nodejs\scripts` directory to update multiple documents.
2. Add an array of documents to the `catalog` collection in the `catalog` database as before using the `insertMany()` method. Add the `catalogId`, `title`, and `author` fields; fields that are unique to all documents, but don't add the common fields `edition`, `journal`, and `publisher`.

```
doc1 = {"catalogId" : 'catalog1', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
doc2 = {"catalogId" : 'catalog2', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
collection.insertMany([doc1, doc2], function(error, result){
if (error)
    console.log(error);
else{
    console.log("Documents added: "+result);
}
});
```

3. Invoke the `updateMany()` method using selection filter to select all documents represented with `{}`. Specify the update using the `$set` update operator to add the common fields `journal`, `publisher` and `edition`. In the callback function, log the method result to the console.

```
collection.updateMany({}, {$set:{edition:'November December 2013',
journal:'Oracle Magazine', publisher:'Oracle Publishing'}},
{upsert:true},
function(error, result) {
if (error) console.log(error);
else
    console.log(result);
});
```

The updateDocuments.js script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1',"title" : 'Engineering as a Service',"author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "title" : 'Quintessential and Collaborative',"author" : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.updateMany({}, {$set:{edition:'November December 2013', journal:'Oracle Magazine', publisher:'Oracle Publishing'}},
        {upsert:true},
        function(error, result) {
          if (error) console.log(error);
          else
            console.log(result);
        });
      }
    });
  });
});
```

4. Before running the script delete the catalog collection from the Mongo shell using the following commands in the Mongo shell.

```
>use local
>db.catalog.drop()
```

5. Run the updateDocuments.js script using the command node updateDocuments.js. Two documents get matched and the two documents get modified as indicated by the nModified field in the result as shown in Figure 5-34.

```
Administrator: C:\Windows\system32\cmd.exe - node updateDocuments.js
C:\Program Files\nodejs\scripts>node updateDocuments.js
Documents added: [Object Object]
{ result: { ok: 1, nModified: 2, n: 2 } }
```

Figure 5-34. Output from updateDocuments.js

- Run the `db.catalog.find()` method in Mongo shell to list the updated documents. Both the documents have the `edition`, `publisher`, and `journal` fields added/updated as shown in Figure 5-35.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
[{"_id": ObjectId("55c7ef050df0a0701fc4955e"), "catalogId": "catalog2", "title": "Quintessential and Collaborative", "author": "Tom Haunert", "edition": "November December 2013", "journal": "Oracle Magazine", "publisher": "Oracle Publishing"}, {"_id": ObjectId("55c7ef050df0a0701fc4955d"), "catalogId": "catalog1", "title": "Engineering as a Service", "author": "David A. Kelly", "edition": "November December 2013", "journal": "Oracle Magazine", "publisher": "Oracle Publishing"}]
```

Figure 5-35. Listing Updated Documents

Removing a Single Document

The `findOneAndDelete(filter, options, callback)` method may be used to remove one or more documents from a collection. The method parameters are discussed in Table 5-25.

Table 5-25. Parameters for the `findOneAndDelete()` Method

Parameter	Type	Description
<code>filter</code>	object	Specifies a document selection filter. If no filter is specified, all documents are removed.
<code>options</code>	object	Method options.
<code>callback</code>	<code>findAndModifyCallback(error, result)</code>	The callback function, which must be specified if using write concern.

The supported options are discussed in Table 5-26.

Table 5-26. Options for the `findOneAndDelete()` Method

Option	Type	Description
<code>projection</code>	object	The fields projection for the result.
<code>sort</code>	object	If the query selects multiple documents, determines which documents are replaced based on the sort order.
<code>maxTimeMS</code>	number	The maximum time in milliseconds for which the query may run.

1. Create a script `removeDocument.js` in the `C:\Program Files\nodejs\scripts` directory.
2. Add an array of documents to a collection using the `insertMany()` method.
3. Invoke the `findOneAndRemove()` method using a selector query to select all documents with journal as Oracle Magazine and output the callback function result to the console.

```
collection.findOneAndRemove({journal:'Oracle Magazine'},
  function(error, result) {
    if (error) console.log(error);
    else
      console.log(result);
  });

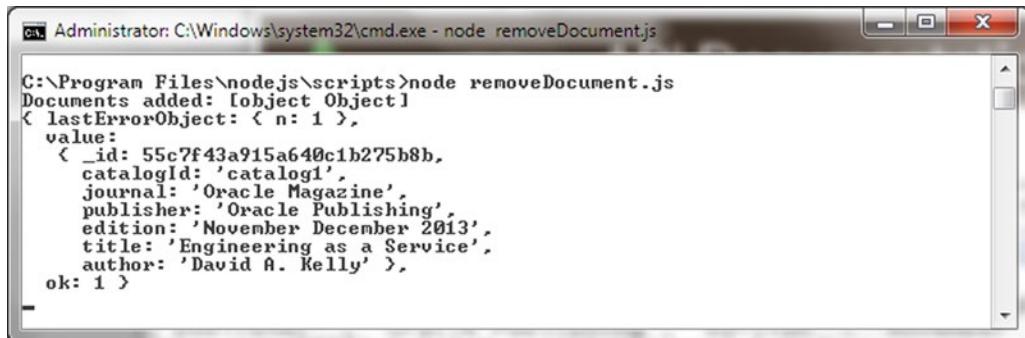
```

The `removeDocument.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Engineering as a Service', "author" : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013', "title" : 'Quintessential and Collaborative', "author" : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.findOneAndDelete({journal:'Oracle Magazine'},
          function(error, result) {
            if (error) console.log(error);
            else
              console.log(result);
            });
      }
    });
  });
});
```

- Drop the catalog collection in local database with db.catalog.drop().

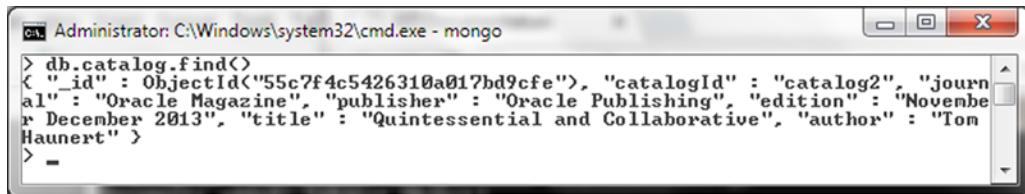
When the removeDocument.js script is run with node removeDocument.js command, two documents get added and one gets removed. The removed document gets output as shown in Figure 5-36.



```
Administrator: C:\Windows\system32\cmd.exe - node removeDocument.js
C:\Program Files\nodejs\scripts>node removeDocument.js
Documents added: [object Object]
{ lastErrorObject: { n: 1 },
  value:
    { _id: 55c7f43a915a640c1b275b8b,
      catalogId: 'catalog1',
      journal: 'Oracle Magazine',
      publisher: 'Oracle Publishing',
      edition: 'November December 2013',
      title: 'Engineering as a Service',
      author: 'David A. Kelly' },
  ok: 1 }
```

Figure 5-36. Output of removeDocument.js

- Run the db.catalog.find() method in Mongo shell to list the documents. As one of the two added documents has been removed, no documents get listed as shown in Figure 5-37.



```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
{ "_id" : ObjectId("55c7f4c5426310a017bd9cfe"), "catalogId" : "catalog2", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013", "title" : "Quintessential and Collaborative", "author" : "Tom Haunert" }
> -
```

Figure 5-37. Listing Documents after Removing One

The deleteOne(filter, options, callback) method is similar to the findOneAndDelete(filter, options, callback) method except that the deleteOne() method supports only the w, wtimeout and j options.

Removing Multiple Documents

The deleteMany(filter, options, callback) method may be used to remove one or more documents from a collection. The method parameters are discussed in Table 5-27.

Table 5-27. Options for the deleteMany() Method

Parameter	Type	Description
filter	object	Specifies a document selection filter.
options	object	Method options. Supported options are w, wtimeout, and j.
callback	writeOpCallback (error, result)	The callback function.

1. Create a script `removeDocuments.js` in the `C:\Program Files\nodejs\scripts` directory.
2. Add an array of documents to a collection using the `insertMany()` method.
3. Invoke the `deleteMany()` method using a selector query to select all documents with `journal` as '`Oracle Magazine`' and output the callback function result to the console.

```
collection.deleteMany({journal:'Oracle Magazine'},function(error,
result) {
  if (error) console.log(error);
  else
    console.log(result);
});
```

The `removeDocuments.js` script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        doc1 = {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" : 'Engineering as a Service','author' : 'David A. Kelly'};
        doc2 = {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" : 'November December 2013',"title" : 'Quintessential and Collaborative','author' : 'Tom Haunert'};
        collection.insertMany([doc1,doc2], function(error, result){
          if (error)
            console.log(error);
          else{
            console.log("Documents added: "+result);
          }
        });
        collection.deleteMany({journal:'Oracle Magazine'},function(error, result) {
          if (error) console.log(error);
          else
            console.log(result);
        });
      }
    });
});
```

- Drop the catalog collection from the local database.

```
>use local
>db.catalog.drop()
```

When the `removeDocuments.js` script is run with the command `node removeDocuments.js` two documents get added and both get removed. The `n` in the result is 2 indicating that two documents have been removed as shown in Figure 5-38.

```
Administrator: C:\Windows\system32\cmd.exe - node removeDocuments.js
C:\Program Files\nodejs\scripts>node removeDocuments.js
Documents added: [object Object]
{ result: { ok: 1, n: 2 } }
```

Figure 5-38. Output of `removeDocuments.js`

- Run the `db.catalog.find()` method in Mongo shell to list the documents. As one of the two added documents has been removed, no documents get listed as shown in Figure 5-39.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
>
> db.catalog.find()
> <
```

Figure 5-39. Listing Documents after Removing All

Performing Bulk Write Operations

The Collection class provides the `bulkWrite(operations, options, callback)` method to perform bulk write operations. The method parameters are discussed in Table 5-28.

Table 5-28. Method Parameters for the `bulkWrite()` Method

Parameter	Type	Description
operations	Array.<object>	Bulk operations to perform. Supported operations are <code>insertOne</code> , <code>updateOne</code> , <code>updateMany</code> , <code>deleteOne</code> , <code>deleteMany</code> , and <code>replaceOne</code> .
options	object	Method options. Supported options are <code>w</code> , <code>wtimeout</code> , <code>j</code> , <code>serializeFunctions</code> , and <code>ordered</code> . The <code>serializeFunctions</code> is <code>false</code> by default. The <code>ordered</code> is <code>true</code> by default, indicating that the write operations are to be performed in order specified.
callback	<code>bulkWriteOpCallback</code>	The callback function.

1. Create a `bulkWriteDocuments.js` script in the `C:\Program Files\nodejs\scripts` directory.
2. Drop the `catalog` collection in the local database.

```
>use local
>db.catalog.drop()
```

3. Create the `catalog` collection in the script and use the `bulkWrite(operations, options, callback)` method to perform bulk write operations to add some documents, update a single document, update multiple documents, replace a document, and delete a document. Set `ordered` option to `true`, which is also the default so that the bulk write operations are performed in the order specified. The order could be significant if a bulk write operation depends on a preceding bulk write operation. For example, documents in a collection cannot be updated before being added.

```
collection.bulkWrite([
  { insertOne: { document: { "catalogId" : 'catalog1', "journal" :
    'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition":
    'November December 2013', "title" : 'Engineering as a Service', "author" :
    'David A. Kelly'} } },
  { insertOne: { document: { "catalogId" : 'catalog2', "journal" :
    'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
    'November December 2013', "title" : 'Quintessential and
    Collaborative', "author" : 'Tom Haunert'} } },
  { insertOne: { document: { "catalogId" : 'catalog3', "journal" :
    'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
    'November December 2013'} } },
  { insertOne: { document: { "catalogId" : 'catalog4', "journal" :
    'Oracle Magazine', "publisher" : 'Oracle Publishing', "edition" :
    'November December 2013'} } }
  , { updateOne: { filter: { journal:'Oracle Magazine'}, update:
    {$set: {journal:'OracleMagazine'}}, upsert:true } }
  , { updateMany: { filter: {edition:'November December 2013'}, update:
    {$set: {edition:'11-12-2013'}}}, upsert:true } }
  , { deleteOne: { filter: {journal:'Oracle Magazine'} } }
  , { replaceOne: { filter: {catalogId:'catalog5'}, replacement:
    {"catalogId" : 'catalog5', "journal" : 'Oracle Magazine',
    "publisher" : 'Oracle Publishing', "edition" : 'November December
    2013'}, upsert:true}}]
  , {ordered:true, w:1}, function(error, result){
  if (error)
    console.log(error);
  else{
    console.log("Documents added: "+result);
  }
});
```

Similarly a { deleteMany: { filter: {journal:'Oracle Magazine'} } } bulk write operation may be added to delete all documents for a selection query. A deleteMany() has not been included in the sample script to demonstrate the effect of the other bulk write operations because if deleteMany() is included, all or most documents would get deleted. The bulkWriteDocuments.js script is listed:

```
Server = require('mongodb').Server;
Db = require('mongodb').Db;
Collection = require('mongodb').Collection;
var db = new Db('local', new Server('localhost', 27017));
db.open(function(error, db) {
  if (error)
    console.log(error);
  else{
    db.createCollection('catalog', function(error, collection){
      if (error)
        console.log(error);
      else{
        collection.bulkWrite([
          { insertOne: { document: {"catalogId" : 'catalog1', "journal" : 'Oracle Magazine',
          "publisher" : 'Oracle Publishing', "edition" :
          'November December 2013','title" : 'Engineering as a Service','author" : 'David A. Kelly'} } },
          { insertOne: { document: {"catalogId" : 'catalog2', "journal" : 'Oracle Magazine',
          "publisher" : 'Oracle Publishing', "edition" :
          'November December 2013','title" : 'Quintessential and Collaborative','author" : 'Tom
          Haunert'} } },
          { insertOne: { document: {"catalogId" : 'catalog3', "journal" : 'Oracle Magazine',
          "publisher" : 'Oracle Publishing', "edition" :
          'November December 2013'} } },
          { insertOne: { document: {"catalogId" : 'catalog4', "journal" : 'Oracle Magazine',
          "publisher" : 'Oracle Publishing', "edition" :
          'November December 2013'} } }
        , { updateOne: { filter: {journal:'Oracle Magazine'}, update: {$set:
          {journal:'OracleMagazine'}}, upsert:true } }
        , { updateMany: { filter: {edition:'November December 2013'}, update: {$set:
          {edition:'11-12-2013'}}}, upsert:true } }
        , { deleteOne: { filter: {journal:'Oracle Magazine'} } }
        , { replaceOne: { filter: {catalogId:'catalog5'}, replacement: {"catalogId" :
          'catalog5', "journal" : 'Oracle Magazine',
          "publisher" : 'Oracle Publishing', "edition" : 'November December 2013'}, upsert:true}}]
        , {ordered:true, w:1}, function(error, result){

      if (error)
        console.log(error);
      else{
        console.log("Documents added: "+result);
      }
    });
  }
});
```

- Run the script with the command `node bulkWriteDocuments.js`. The output is shown in Figure 5-40.

```
Administrator: C:\Windows\system32\cmd.exe - node bulkWriteDocuments.js
C:\Program Files\nodejs\scripts>node bulkWriteDocuments.js
Documents added: BulkWriteResult<1 object Object>
```

Figure 5-40. Running the `bulkWriteDocuments.js` Script

- Subsequently run the `db.catalog.find()` command in Mongo shell to list the documents as shown in Figure 5-41.

```
Administrator: C:\Windows\system32\cmd.exe - mongo
> db.catalog.find()
{ "_id" : ObjectId("55c7ffe0f5e17bfc15988892"), "catalogId" : "catalog1", "journal" : "OracleMagazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013", "title" : "Engineering as a Service", "author" : "David A. Kelly" }
{ "_id" : ObjectId("55c7ffe0f5e17bfc15988894"), "catalogId" : "catalog3", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013" }
{ "_id" : ObjectId("55c7ffe0f5e17bfc15988895"), "catalogId" : "catalog4", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "11-12-2013" }
{ "_id" : ObjectId("55c7ffe0ee59f736d1f6b98d"), "catalogId" : "catalog5", "journal" : "Oracle Magazine", "publisher" : "Oracle Publishing", "edition" : "November December 2013" }
> -
```

Figure 5-41. Listing Documents

As shown in the output only four documents are listed because four were added initially and one was deleted and subsequently a document was upserted. One of the documents has journal set to OracleMagazine because `updateOne` was used to update the journal field of one document. The catalog5 catalogId document is the document upserted with `replaceOne`.

Summary

In this chapter we used the Node.js driver for MongoDB to connect to MongoDB server and perform CRUD (create, read, update, and delete) operations. We demonstrated the CRUD operations for single and multiple documents. In the next chapter we shall migrate a document from Apache Cassandra to MongoDB.

CHAPTER 6



Migrating an Apache Cassandra Table to MongoDB

While MongoDB is the most commonly used document-based NoSQL datastore, Apache Cassandra is the most commonly used wide column-based NoSQL datastore. The data models used in MongoDB and Apache Cassandra are different. MongoDB is based on a JSON (BSON) data model while Cassandra is based on a column/row (table) data model. Both provide a schema-less, flexible storage model. In this chapter we shall discuss migrating Apache Cassandra documents to MongoDB. This chapter includes the following topics:

- Setting Up the Environment
- Creating a Maven Project in Eclipse
- Creating a Document in Apache Cassandra
- Migrating the Cassandra Table to MongoDB

Setting Up the Environment

We need to install the following software for this chapter:

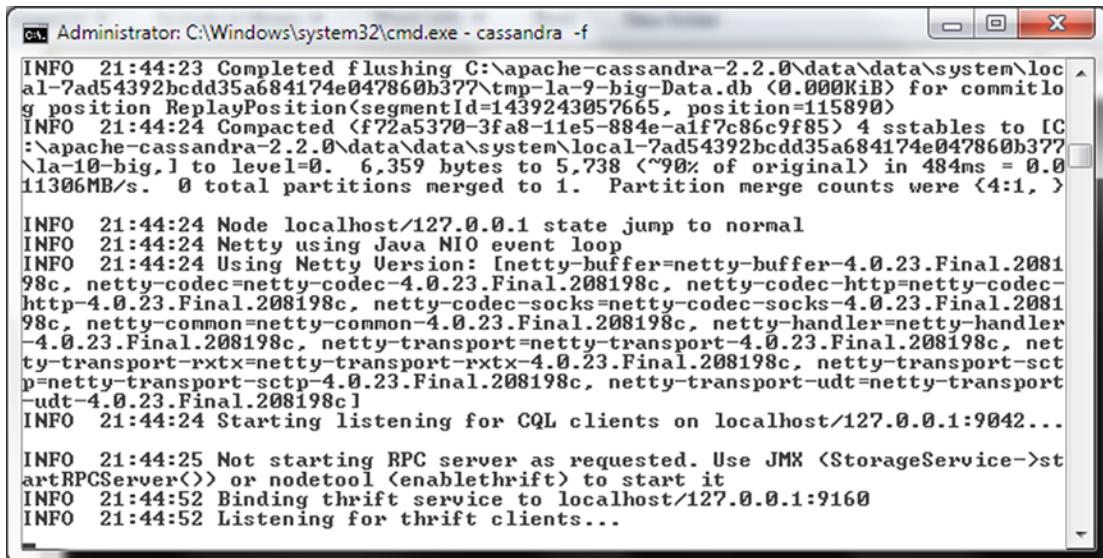
- Apache Cassandra 2.2.0 apache-cassandra-2.2.0-bin.tar.gz from <http://cassandra.apache.org/download/>. Extract tar.gz file to a directory and add the bin directory, for example, the C:\apache-cassandra-2.2.0\bin directory to the PATH variable.
- MongoDB 3.0.5 from www.mongodb.org/downloads.
- Eclipse IDE for Java EE Developers from www.eclipse.org/downloads.
- Java 7 from www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Double-click on the MongoDB binary distribution to install MongoDB. Add the bin directory (C:\Program Files\MongoDB\Server\3.0\bin) of the MongoDB installation to the PATH environment. Create a directory C:\data\db for the MongoDB data if not already created for an earlier chapter.

Start Apache Cassandra server with the following command.

```
>cassandra -f
```

The server gets started as shown in the server output in Figure 6-1. If the thrift service does not get started, run the command `nodetool enablethrift`. The Cassandra server is listening for clients on localhost on port 9160.



A screenshot of a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - cassandra -f". The window contains the following log output:

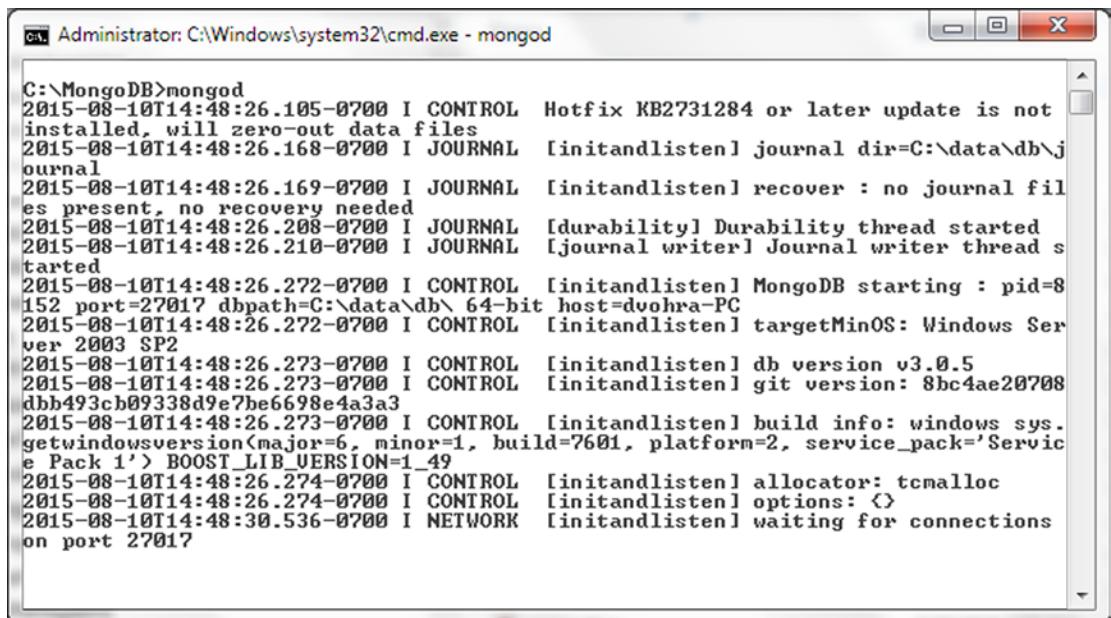
```
INFO 21:44:23 Completed flushing C:\apache-cassandra-2.2.0\data\data\system\local-7ad54392bcd35a684174e047860b377\tmp-la-9-big>Data.db <0.000KiB> for commitlog position ReplayPosition(segmentId=1439243057665, position=115890)
INFO 21:44:24 Compacted <f72a5370-3fa8-11e5-884e-a1f7c86c9f85> 4 sstables to [C:\apache-cassandra-2.2.0\data\data\system\local-7ad54392bcd35a684174e047860b377\la-10-big,] to level=0. 6,359 bytes to 5,738 (~90% of original) in 484ms = 0.011306MB/s. 0 total partitions merged to 1. Partition merge counts were {4:1, }
INFO 21:44:24 Node localhost/127.0.0.1 state jump to normal
INFO 21:44:24 Netty using Java NIO event loop
INFO 21:44:24 Using Netty Version: [netty-buffer=netty-buffer-4.0.23.Final.208198c, netty-codec=netty-codec-4.0.23.Final.208198c, netty-codec-http=netty-codec-http-4.0.23.Final.208198c, netty-codec-socks=netty-codec-socks-4.0.23.Final.208198c, netty-common=netty-common-4.0.23.Final.208198c, netty-handler=netty-handler-4.0.23.Final.208198c, netty-transport=netty-transport-4.0.23.Final.208198c, netty-transport-rxtx=netty-transport-rxtx-4.0.23.Final.208198c, netty-transport-sctp=netty-transport-sctp-4.0.23.Final.208198c, netty-transport-udt=netty-transport-udt-4.0.23.Final.208198c]
INFO 21:44:24 Starting listening for CQL clients on localhost/127.0.0.1:9042...
INFO 21:44:25 Not starting RPC server as requested. Use JMX <StorageService->startRPCServer() or nodetool {enablethrift} to start it
INFO 21:44:52 Binding thrift service to localhost/127.0.0.1:9160
INFO 21:44:52 Listening for thrift clients...
```

Figure 6-1. Starting Apache Cassandra

Start the MongoDB server with the following command.

```
>mongod
```

The MongoDB server gets started as shown in Figure 6-2. The MongoDB is waiting for connections on localhost on port 27017.



The screenshot shows a Windows Command Prompt window titled 'Administrator: C:\Windows\system32\cmd.exe - mongod'. The command entered is 'mongod'. The output log shows the MongoDB server starting up, including initialization messages, journaling details, and system information like the operating system version (Windows Server 2003 SP2) and build details (db version v3.0.5, git version: 8bc4ae20708). It also mentions the use of the BOOST_LIB_VERSION=1_49 library.

```
C:\MongoDB>mongod
2015-08-10T14:48:26.105-0700 I CONTROL  Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-08-10T14:48:26.168-0700 I JOURNAL  [initandlisten] journal dir=C:\data\db\j
ournal
2015-08-10T14:48:26.169-0700 I JOURNAL  [initandlisten] recover : no journal fil
es present, no recovery needed
2015-08-10T14:48:26.208-0700 I JOURNAL  [durability] Durability thread started
2015-08-10T14:48:26.210-0700 I JOURNAL  [journal writer] Journal writer thread s
tarted
2015-08-10T14:48:26.272-0700 I CONTROL  [initandlisten] MongoDB starting : pid=8
152 port=27017 dbpath=C:\data\db\ 64-bit host=dvohra-PC
2015-08-10T14:48:26.272-0700 I CONTROL  [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-08-10T14:48:26.273-0700 I CONTROL  [initandlisten] db version v3.0.5
2015-08-10T14:48:26.273-0700 I CONTROL  [initandlisten] git version: 8bc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-08-10T14:48:26.273-0700 I CONTROL  [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-08-10T14:48:26.274-0700 I CONTROL  [initandlisten] allocator: tcmalloc
2015-08-10T14:48:26.274-0700 I CONTROL  [initandlisten] options: <>
2015-08-10T14:48:30.536-0700 I NETWORK  [initandlisten] waiting for connections
on port 27017
```

Figure 6-2. Starting MongoDB Server

Creating a Maven Project in Eclipse

Next, create a Java project in Eclipse IDE for migrating Cassandra database data to MongoDB database.

1. Select File ▶ New ▶ Other.
2. In the New window, select Maven ▶ Maven Project and click on Next as shown in Figure 6-3.

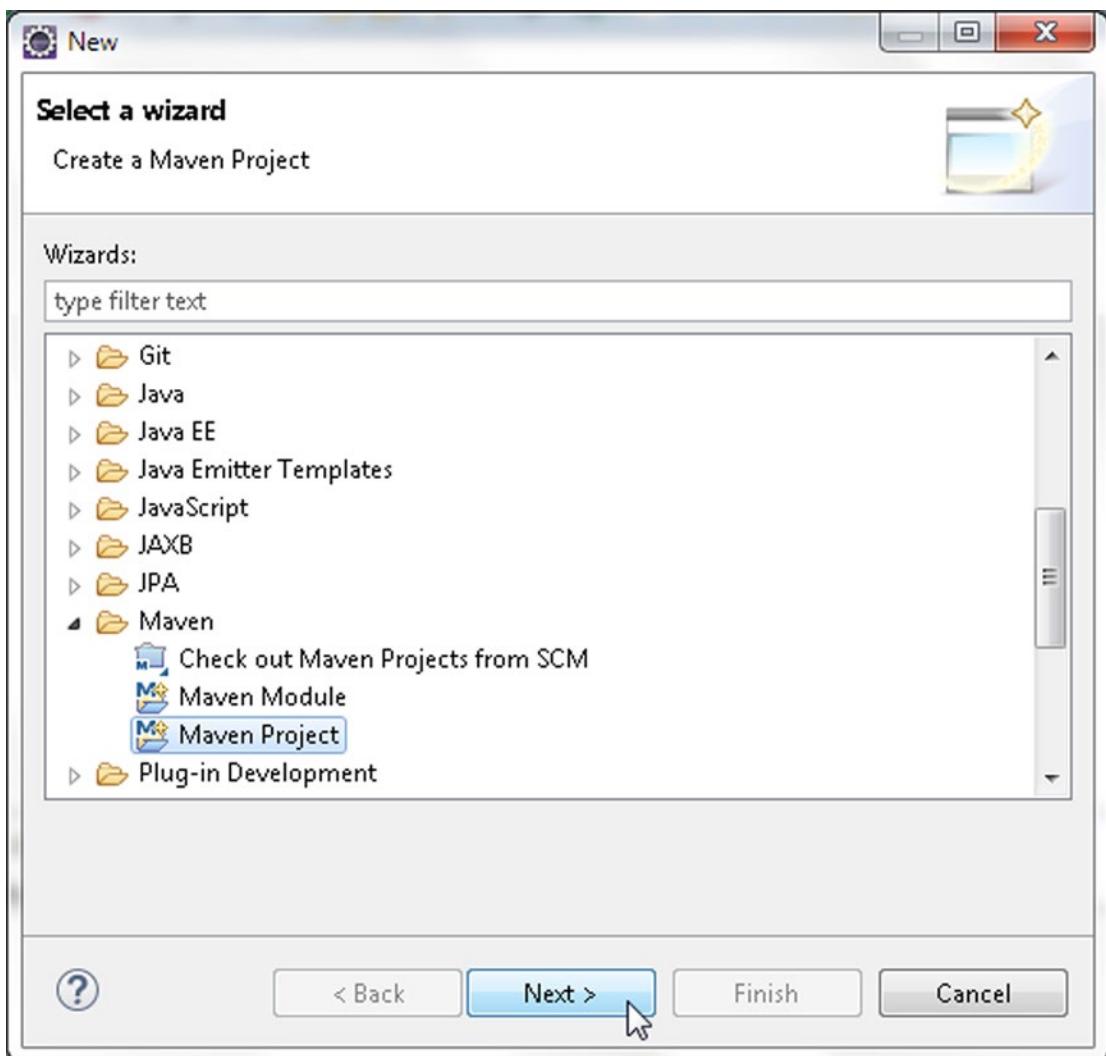


Figure 6-3. Selecting **Maven > Maven Project**

3. The New Maven Project wizard gets started. Select the Create a simple project check box and the Use default Workspace location check box and click on Next as shown in Figure 6-4.

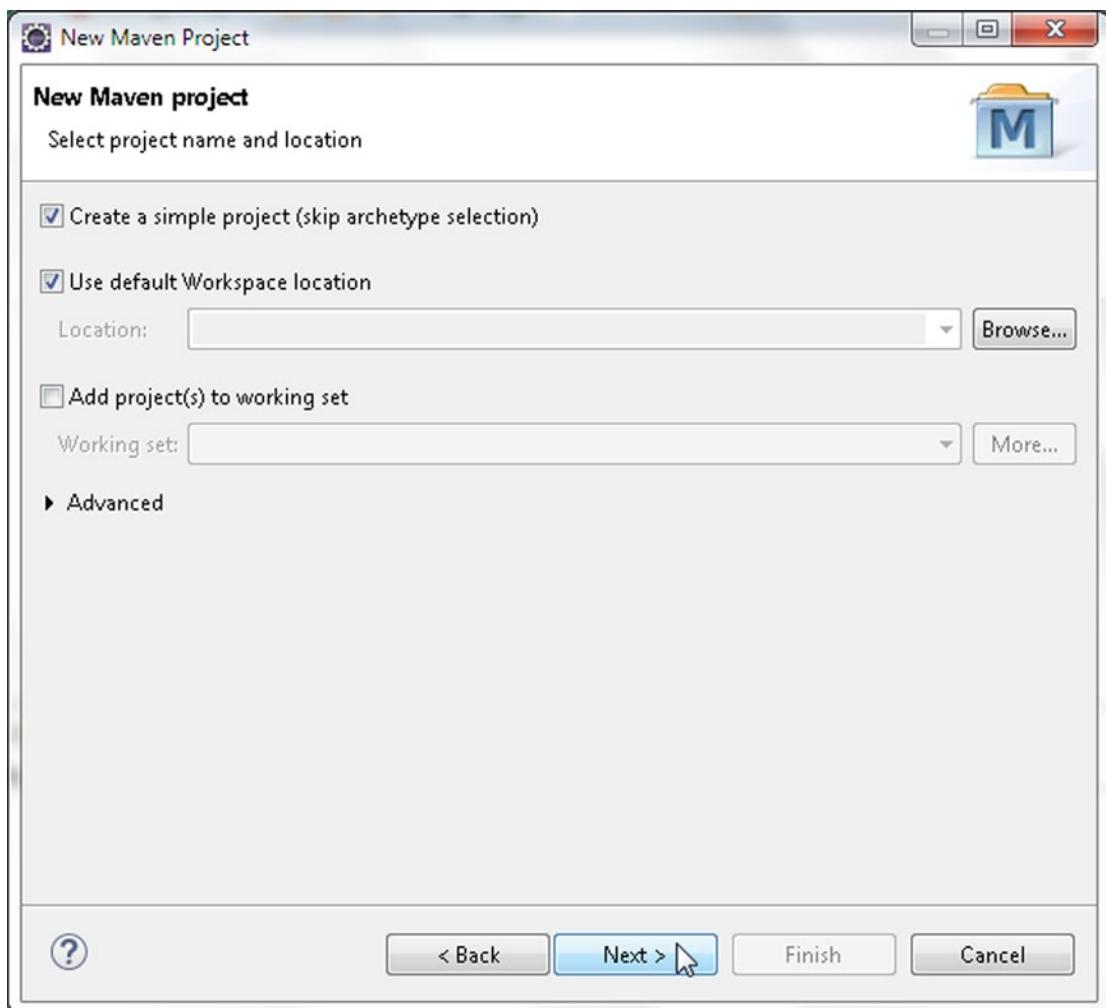


Figure 6-4. New Maven Project Wizard

4. In Configure project, specify the following and then click on Finish as shown in Figure 6-5.
 - Group Id: com.mongodb.migration
 - Artifact Id: CassandraToMongoDB
 - Version: 1.0.0
 - Packaging: jar
 - Name: CassandraToMongoDB

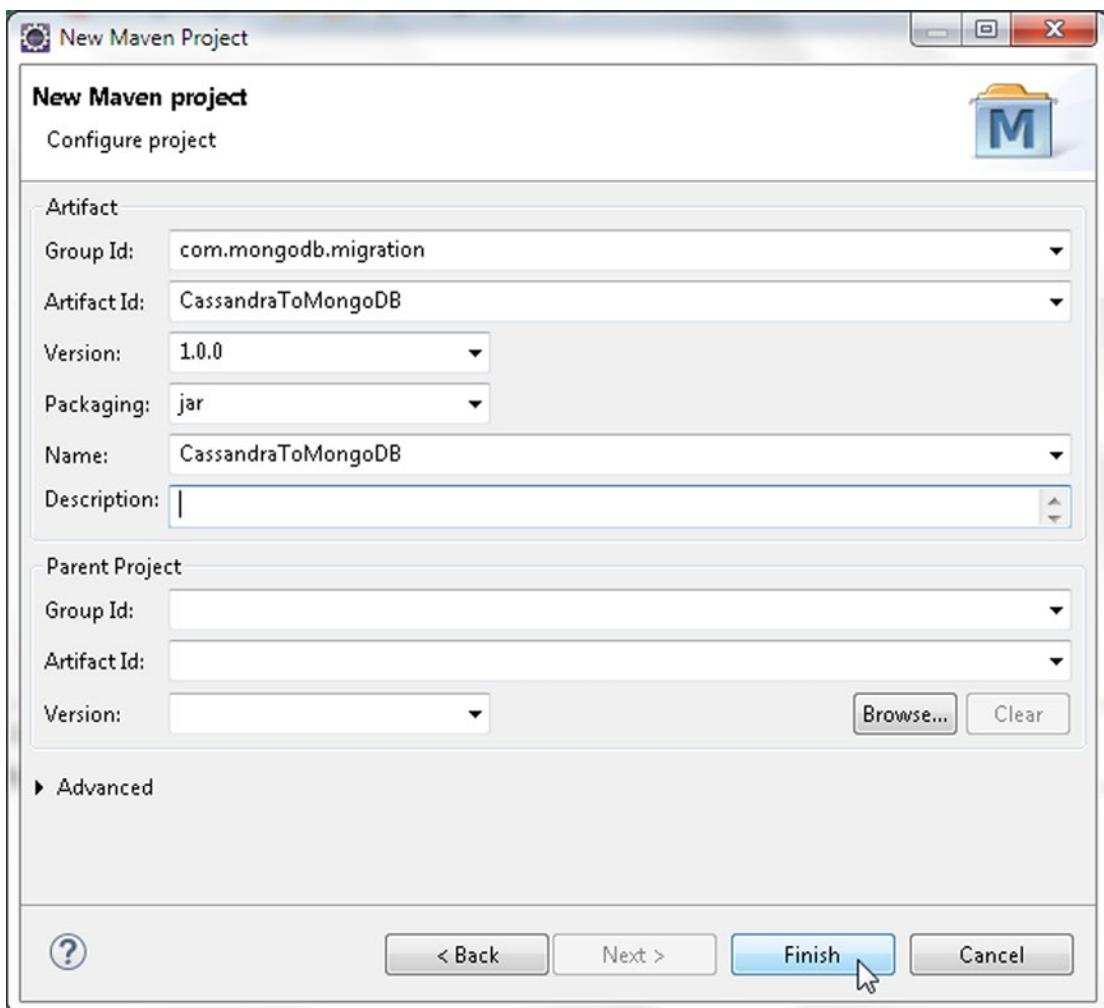


Figure 6-5. Configuring Maven Project

A Maven Project gets created in Eclipse IDE as shown in Figure 6-6.

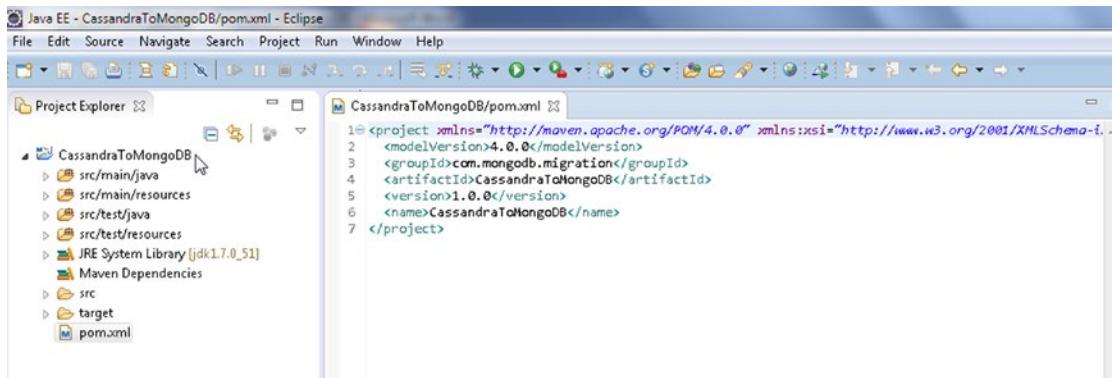


Figure 6-6. Maven Project *CassandraToMongoDB* in Package Explorer

Now we need to create two Java classes for the migration: one to create the initial data in Cassandra and the other to migrate the data to MongoDB.

1. To create a Java class click on File ▶ New ▶ Other.
2. In the New window, select Java ▶ Class and click on Next as shown in Figure 6-7.

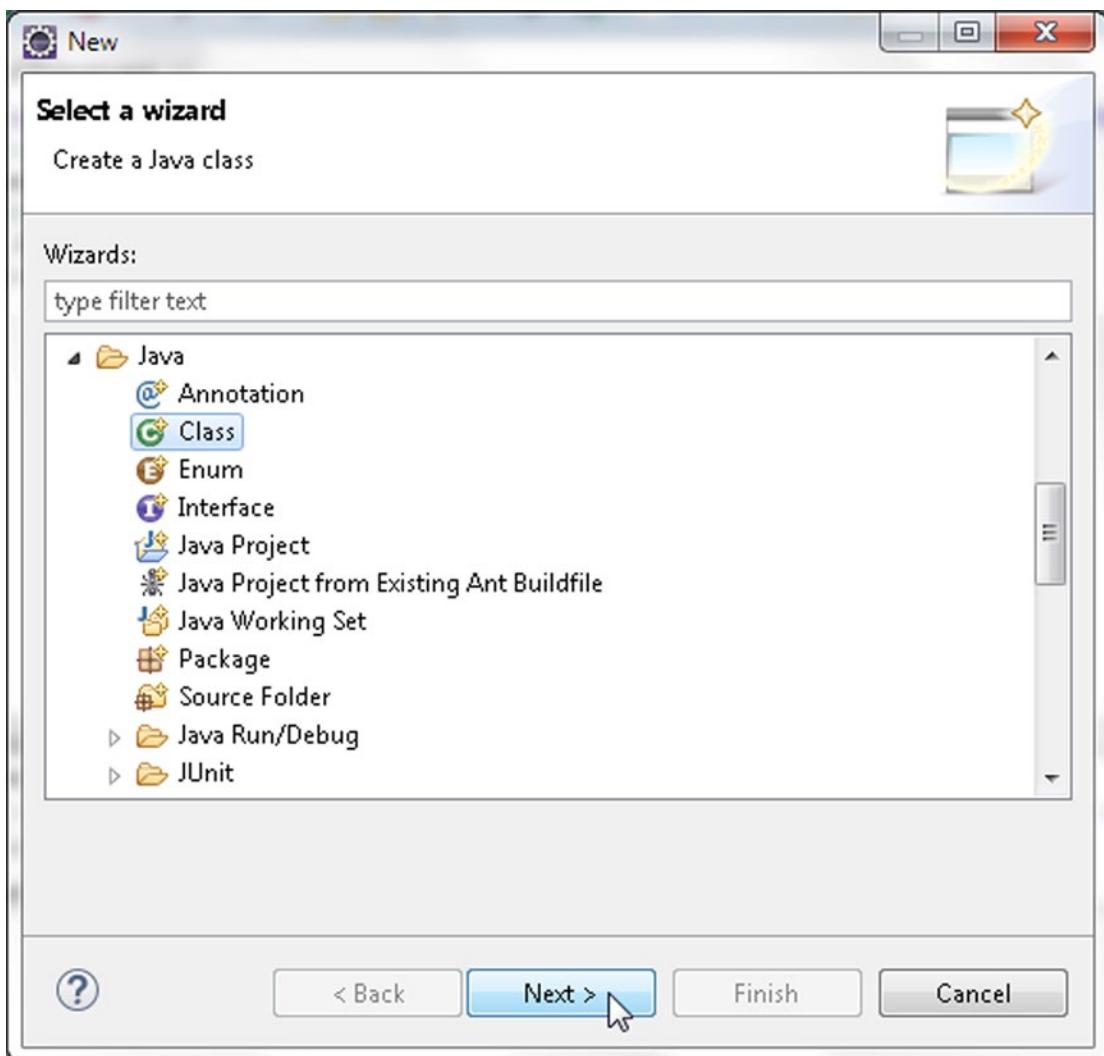


Figure 6-7. Selecting Java ► Java Class

3. In New Java Class wizard select the Source folder as CassandraToMongoDB/src/main/java and specify Package as mongodb and class name as CreateCassandraDatabase. Select the check box to create method stub public static void main(String[] args). Click on Finish as shown in Figure 6-8.

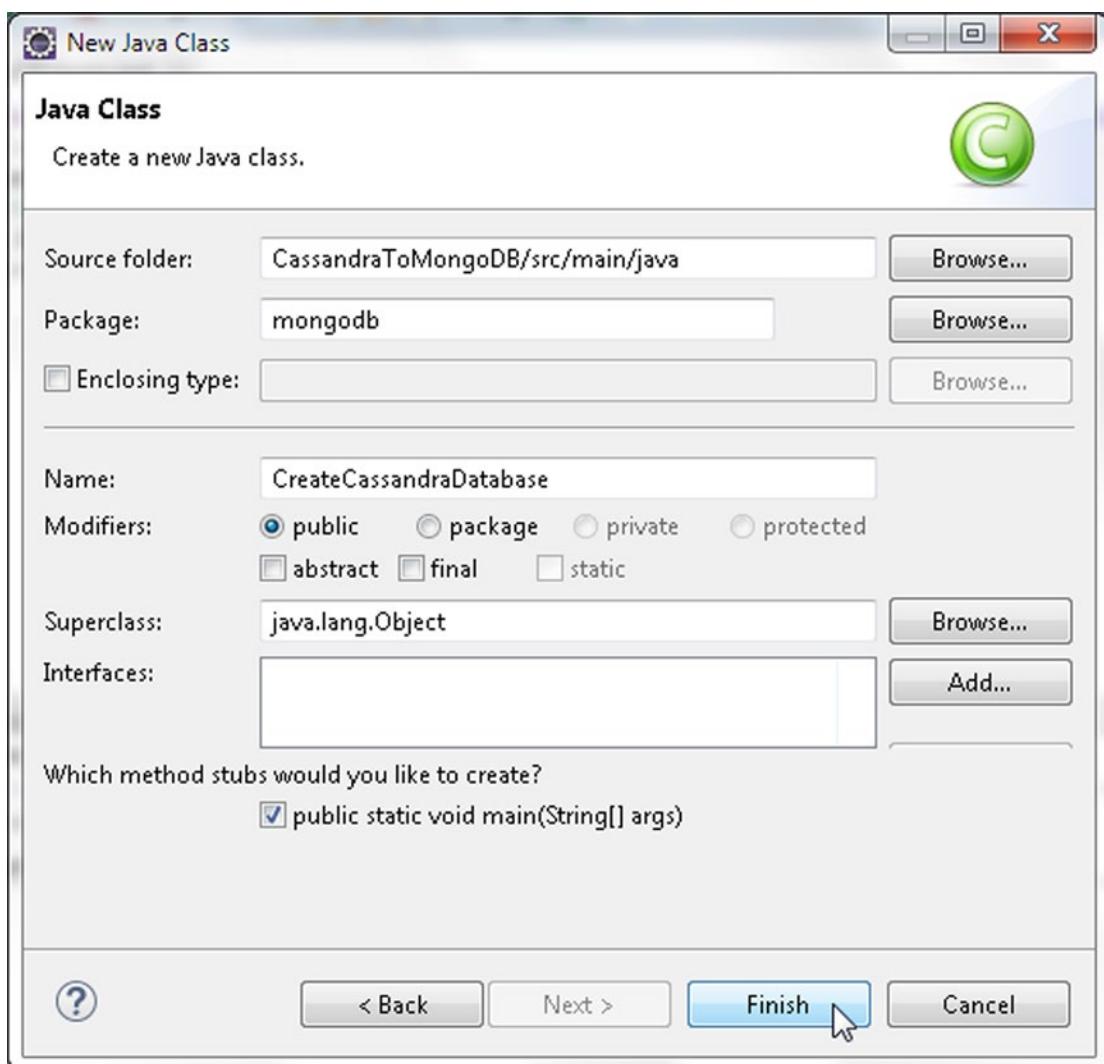


Figure 6-8. Configuring Java Class CreateCassandraDatabase

4. Similarly create a Java class MigrateCassandraToMongoDB as shown in Figure 6-9.

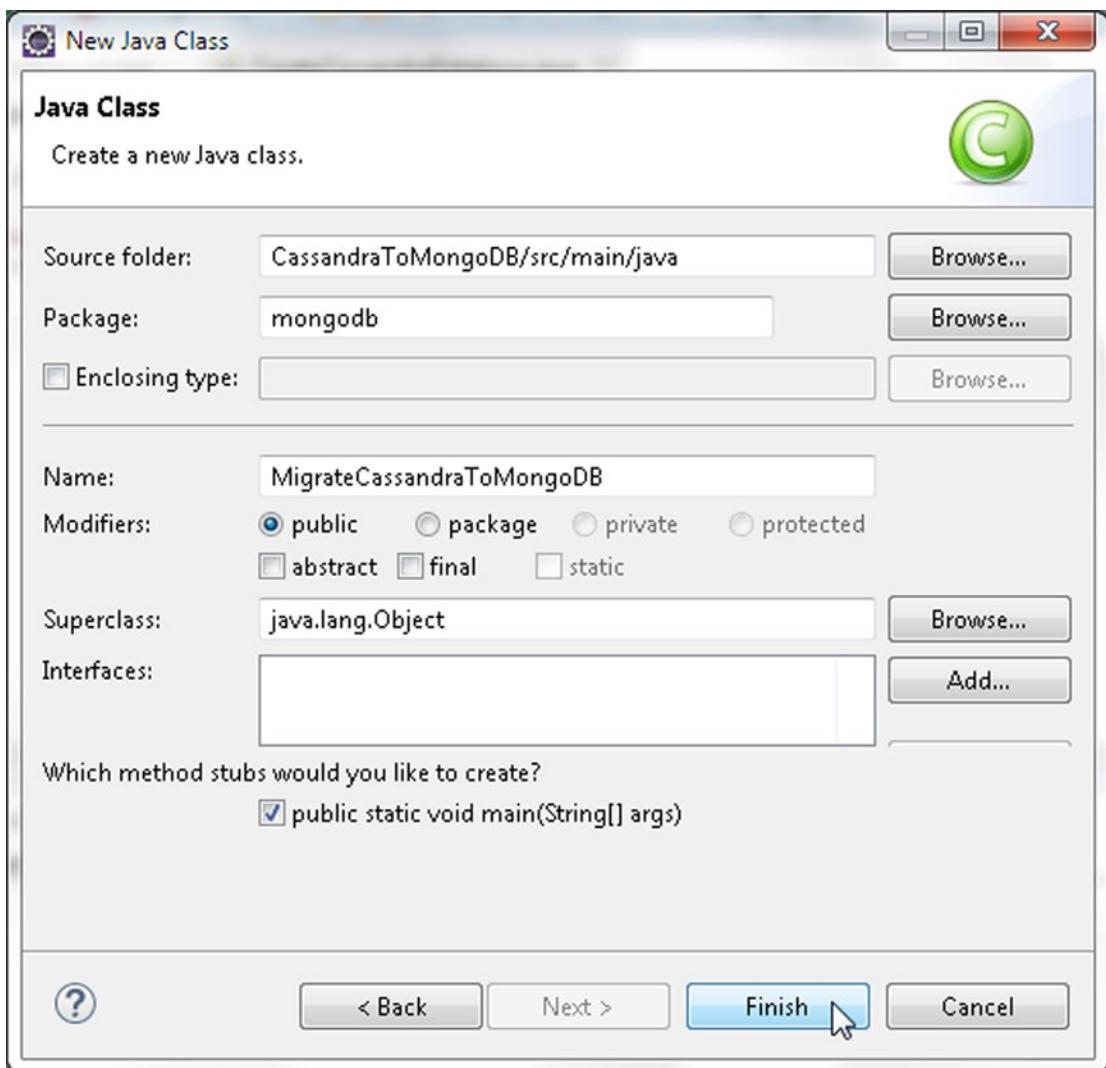


Figure 6-9. Configuring Java Class MigrateCassandraToMongoDB

The two Java classes are shown in the Package Explorer in Figure 6-10.

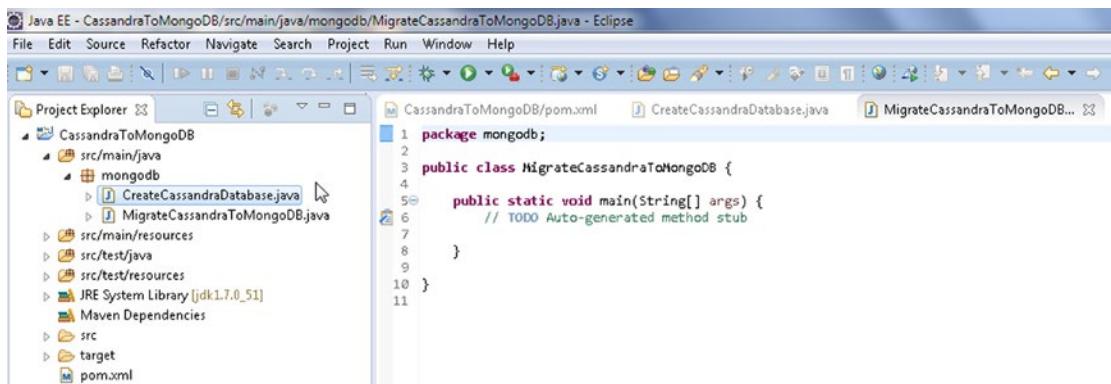


Figure 6-10. Java Classes in Package Explorer

We need to add some dependencies to the `pom.xml`. Add the following dependencies listed in Table 6-1; some of the dependencies are indicated as being included with the Apache Cassandra Project dependency and should not be added separately.

Table 6-1. Dependencies

Jar	Description
Mongo Java Driver 3.0.3	The Java Client to MongoDB Server.
Apache Cassandra 2.2.0	Apache Cassandra Project.
Cassandra Driver Core 2.2.0-rc2	The Datastax Java driver.
Apache Commons BeanUtils 1.9.2	Utility Jar for Java classes developed with the JavaBeans pattern.
Apache Commons Collections 3.2.1	Provides data structures that accelerate Java application development.
Apache Commons Lang 3 3.1	Provides extra classes for manipulation of Java core classes. Included with Apache Cassandra Project dependency.
Apache Commons Logging 1.2	An interface for common logging implementations.
Guava 16.0.1	Google's core libraries used in Java-based projects. Included with Apache Cassandra dependency.
Jackson Core ASL 1.9.2	High-performance JSON processor. Included with Apache Cassandra Project dependency.
Jackson Mapper ASL 1.9.2	High-performance data binding package built on Jackson JSON processor. Included with Apache Cassandra Project dependency.
Metrics Core 3.1.0	The core library for Metrics. Included with Apache Cassandra Project dependency.
Netty 4.0.27	NIO client server framework to develop network applications such as protocol servers and clients. Included with Apache Cassandra Project dependency.
Slf4j API 1.7.7	Simple Logging Façade for Java, which serves as an abstraction for various logging frameworks. Included with Apache Cassandra Project dependency.

The pom.xml is listed below.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mongodb.migration</groupId>
  <artifactId>CassandraToMongoDB</artifactId>
  <version>1.0.0</version>
  <name>CassandraToMongoDB</name>

  <dependencies>
    <dependency>
      <groupId>org.mongodb</groupId>
      <artifactId>mongo-java-driver</artifactId>
      <version>3.0.3</version>
    </dependency>

    <dependency>
      <groupId>com.datastax.cassandra</groupId>
      <artifactId>cassandra-driver-core</artifactId>
      <version>2.2.0-rc2</version>
    </dependency>

    <dependency>
      <groupId>org.apache.cassandra</groupId>
      <artifactId>cassandra-all</artifactId>
      <version>2.2.0</version>
    </dependency>

    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.9.2</version>
    </dependency>

    <dependency>
      <groupId>commons-collections</groupId>
      <artifactId>commons-collections</artifactId>
      <version>3.2.1</version>
    </dependency>

    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>

</project>
```

Some of these dependencies have further dependencies, which get added automatically and should not be added separately. To find the required Jars that get added from the dependencies, right-click on the project node in Package Explorer and select Properties. In Properties select Java Build Path. The Jars added to the migration project are shown in Figure 6-11.

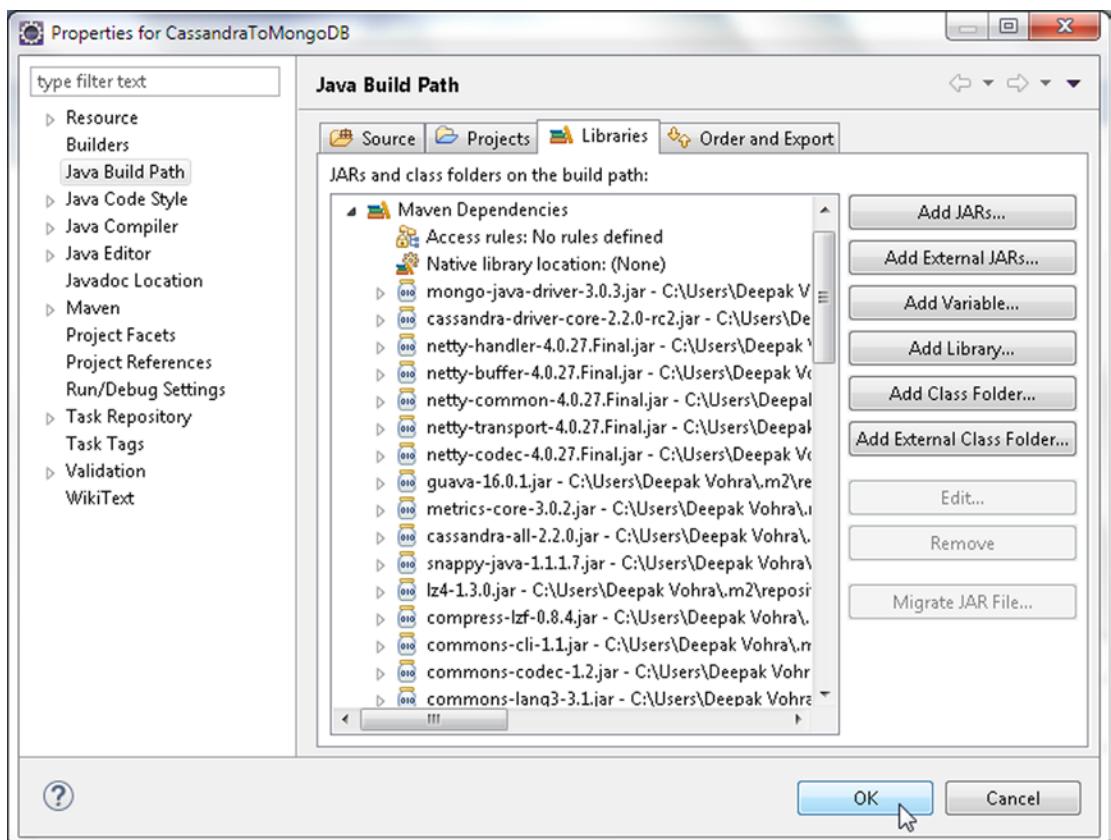


Figure 6-11. Jar Files in the Java Build Path

Creating a Document in Apache Cassandra

In this section we shall create the Cassandra table to be migrated to MongoDB. A Cassandra table may be created either using the Cassandra-CLI or using a Java application with Cassandra Java driver. We shall create a Cassandra table in a Java application, the CreateCassandraDatabase application. In the Java application, first we need to connect to Cassandra from the application. We shall use the Datastax Java driver to connect to Cassandra. Create an instance of Cluster, which is the main entry point for the Datastax Java driver. The cluster maintains a connection with one of the server nodes to keep information on the state and current topology of the cluster. The driver discovers all the nodes in the cluster making use of auto-discovery of nodes, which includes new nodes that join later. Build a Cluster.Builder instance, which is a helper class to build Cluster instances, using the static method builder().

1. We need to provide the connect address of at least one of the nodes in the Cassandra cluster for the Datastax driver to be able to connect with the cluster and discover other nodes in the cluster using auto-discovery. Add the address of the Cassandra server running on the localhost (127.0.0.1) using the `addContactPoint(String)` method of `Cluster.Builder`.
2. Next, invoke the `build()` method to build the `Cluster` using the configured addresses. The methods may be invoked in sequence as we don't need an instance of the intermediary `Cluster.Builder`.

```
cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
```

3. Next, create a session on the cluster by invoking the `connect()` method. A `Session` instance is used to query the cluster and is represented with the `Session` class, which holds multiple connections to the cluster. The `Session` instance also provides policies on which node in the cluster to use for querying the cluster. The default policy is to use a round robin on all the nodes in the cluster. `Session` is also used to handle retries of failed queries. `Session` instances are thread-safe and a single instance is sufficient for an application if connecting to a single keyspace only. A separate `Session` instance is required if connecting to multiple keyspaces.

```
Session session = cluster.connect();
```

The Cassandra server must be running to be able to connect to the server when the application is run, and we already started the Cassandra server earlier. If Cassandra server is not running, the `com.datastax.driver.core.exceptions.NoHostAvailableException` exception is generated when a connection is tried.

The `Session` class provides several methods to prepare and run queries on the server, some of which are discussed in Table 6-2.

Table 6-2. Session Class Methods to Run Queries

Method	Description
<code>execute(Statement statement)</code>	Executes the query provided as a <code>Statement</code> object to return a <code>ResultSet</code> .
<code>execute(String query)</code>	Executes the query provided as a <code>String</code> to return a <code>ResultSet</code> .
<code>execute(String query, Object... values)</code>	Executes the query provided as a <code>String</code> and uses the specified <code>values</code> to return a <code>ResultSet</code> .

4. We need to create a keyspace to store tables in. Add a static method `createKeyspace()` to create a keyspace in the `CreateCassandraDatabase` application. CQL 3 (Cassandra Query Language 3) has added support to run `CREATE` statements conditionally, which implies that an object is created only if the object to be constructed does not already exist. The `IF NOT EXISTS` clause is used to create conditionally. Create a keyspace called `datastax` using replication with strategy class as `SimpleStrategy` and replication factor as 1.

```
session.execute("CREATE KEYSPACE IF NOT EXISTS datastax WITH replication "
+ "= {'class':'SimpleStrategy', 'replication_factor':1};");
```

5. Invoke the `createKeyspace()` method in the `main` method. When the application is run, a keyspace gets created. Cassandra supports the following strategy classes listed in Table 6-3 that refer to the replica placement strategy class.

Table 6-3. Strategy Classes

Class	Description
<code>org.apache.cassandra.locator.SimpleStrategy</code>	Used for a single data center only. The first replica is placed on a node as determined by the partitioner. Subsequent replicas are placed on the next node/s in a clockwise manner in the ring of nodes without consideration to topology. The replication factor is required only if <code>SimpleStrategy</code> class is used.
<code>org.apache.cassandra.locator.NetworkTopologyStrategy</code>	Used with multiple data centers. Specifies how many replicas to store in each data center. Attempts to store replicas on different racks within the same data center because nodes in the same rack are more likely to fail together.

6. Next, we shall create a column family, which is also called a table in CQL 3. Add a static method `createTable()` to `CreateCassandraDatabase` application. As mentioned before `CREATE TABLE` command also supports `IF NOT EXISTS` to create a table conditionally. CQL 3 has added the provision to create a compound primary key, a primary key created from multiple component primary key columns. In a compound primary key the first column is called the partition key. Create a table called `catalog`, which has columns `catalog_id`, `journal`, `publisher`, `edition`, `title`, and `author`. In `catalog` table the compound primary key is made from `catalog_id` and `journal` columns with `catalog_id` being the partition key. Invoke the `execute(String)` method to create table `catalog` as follows.

```
session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog (catalog_id text,journal text,publisher text, edition text,title text,author text,PRIMARY KEY (catalog_id, journal))");
```

7. Prefix the table name with the keyspace name. Invoke the `createTable()` method in `main` method. When the `CreateCassandraDatabase` application is run, the `catalog` table gets created.
8. Next, we shall add data to the table `catalog` using the `INSERT` statement. Use the `IF NOT EXISTS` keyword to add rows conditionally. When a compound primary key is used, all the component primary key columns must be specified including the values for the compound key columns.
9. Add a method `insert()` to the `CreateCassandraDatabase` class and invoke the method in the `main()` method.
10. Add two rows identified by row ids `catalog1`, `catalog2` to the table `catalog`. For example, the two rows are added to the `catalog` table as follows.

```
session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher, edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing', 'November-December 2013', 'Engineering as a Service','David A. Kelly') IF NOT EXISTS");
```

```
session.execute("INSERT INTO datastax.catalog (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog2','Oracle
Magazine', 'Oracle Publishing', 'November-December 2013',
'Quintessential and Collaborative','Tom Haunert') IF NOT EXISTS");
```

11. To verify that a Cassandra table got created, next we shall run a SELECT statement to select columns from the catalog table. Add a method select() to run SELECT statement/s. Select all the columns from the catalog table using the * for column selection. The SELECT statement is run as a test to find that the data we added actually did get added.

```
ResultSet results = session.execute("select * from datastax.catalog");
```

12. A row in the ResultSet is represented with the Row class. Iterate over the ResultSet to output the column value or each of the columns.

```
for (Row row : results) {
    System.out.println("Catalog Id: " + row.getString("catalog_id"));
    System.out.println("\n");
    System.out.println("Journal: " + row.getString("journal"));
    System.out.println("Publisher: " + row.
        getString("publisher"));
    System.out.println("Edition: " + row.getString("edition"));
    System.out.println("Title: " + row.getString("title"));
    System.out.println("Author: " + row.getString("author"));
    System.out.println("\n");
    System.out.println("\n");
}
```

The CreateCassandraDatabase class is listed below.

```
package mongodb;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;

public class CreateCassandraDatabase {
    private static Cluster cluster;
    private static Session session;

    public static void main(String[] argv) {
        cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
        session = cluster.connect();
        createKeyspace();
        createTable();
        insert();
        select();
        session.close();
        cluster.close();
    }
}
```

```

private static void createKeyspace() {
    session.execute("CREATE KEYSPACE IF NOT EXISTS datastax WITH replication "
        + "= {'class':'SimpleStrategy', 'replication_factor':1};");
}

private static void createTable() {
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog (catalog_id text,journal
text,publisher text, edition text,title text,author text,PRIMARY KEY (catalog_id, journal))");
}

private static void insert() {
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service','David A. Kelly') IF NOT EXISTS");
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative','Tom Haunert') IF NOT EXISTS");
}

private static void select() {
    ResultSet results = session.execute("select * from datastax.catalog");
    for (Row row : results) {
        System.out.println("Catalog Id: " + row.getString("catalog_id"));
        System.out.println("\n");
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("\n");
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("\n");
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("\n");
        System.out.println("Title: " + row.getString("title"));
        System.out.println("\n");
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
    }
}
}

```

13. Run the `CreateCassandraDatabase` application to add two rows of data to the catalog table. Right-click on `CreateCassandraDatabase.java` in Package Explorer and select Run As ➤ Java Application as shown in Figure 6-12.

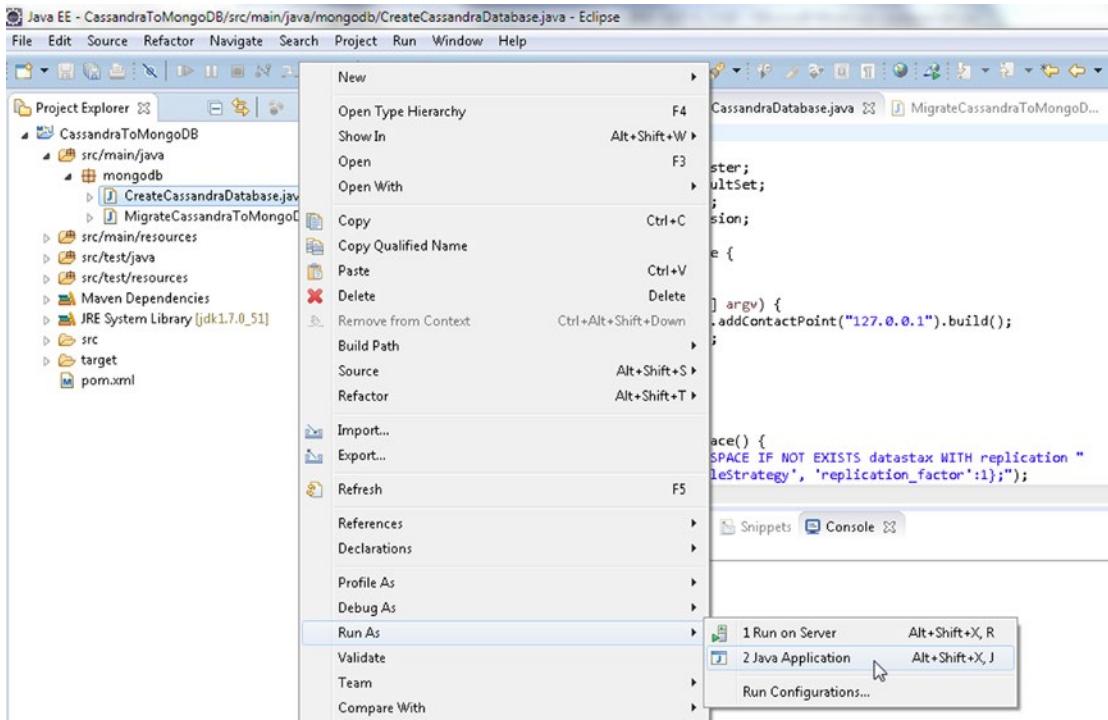


Figure 6-12. Running Java Application *CreateCassandraDatabase.java*

The Cassandra keyspace `datastax` gets created, the catalog table gets created, and data gets added to the table. The `SELECT` statement, which is run as a test, outputs the two rows added to Cassandra as shown in Figure 6-13.

The screenshot shows a Java application window titled "CreateCassandraDatabase [Java Application]". The window has tabs at the top: Properties, Servers, Data Source Explorer, Snippets, and Console. The Data Source Explorer tab is selected. The console output shows two rows of data being added to a Cassandra database:

```
CreateCassandraDatabase [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 10, 2015, 3:38:51 PM)
Catalog Id: catalog1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
Catalog Id: catalog2
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert
```

Figure 6-13. The Two Rows of Data Added to Cassandra Database

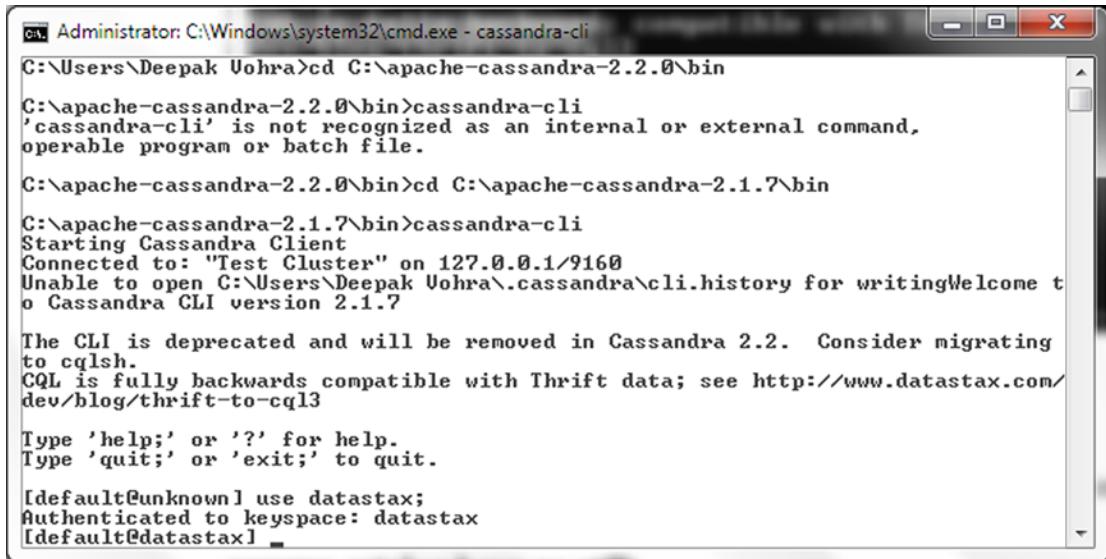
14. To verify that the `datastax` keyspace got created in Cassandra, log in to the Cassandra Client interface with the following command. If the Apache Cassandra version used does not include a `cassandra-cli`, use an earlier version Apache Cassandra 2.1.7 for the `cassandra-cli`.

```
cassandra-cli
```

15. Run the following command to authenticate the `datastax` keyspace.

```
use datastax;
```

The datastax keyspace gets authenticated as shown in Figure 6-14.



A screenshot of a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - cassandra-cli". The command line shows the user navigating through directory paths to reach the Cassandra-CLI bin folder. An error message indicates that 'cassandra-cli' is not recognized as an internal or external command. The user then switches to a newer version of the CLI (2.1.7) and connects to a "Test Cluster" at port 9160. A warning message notes that the CLI is deprecated and will be removed in Cassandra 2.2. The user is prompted to use cqlsh instead. The CLI is described as fully backwards compatible with Thrift data. The user then selects the "datastax" keyspace, which is authenticated successfully. The session ends with a prompt for help or quit.

```
Administrator: C:\Windows\system32\cmd.exe - cassandra-cli
C:\Users\Deepak Vohra>cd C:\apache-cassandra-2.2.0\bin
C:\apache-cassandra-2.2.0\bin>cassandra-cli
'cassandra-cli' is not recognized as an internal or external command,
operable program or batch file.

C:\apache-cassandra-2.2.0\bin>cd C:\apache-cassandra-2.1.7\bin
C:\apache-cassandra-2.1.7\bin>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Unable to open C:\Users\Deepak Vohra\.cassandra\cli.history for writingWelcome to Cassandra CLI version 2.1.7

The CLI is deprecated and will be removed in Cassandra 2.2. Consider migrating to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/dev/blog/thrift-to-cql3

Type 'help;' or '??' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown] use datastax;
Authenticated to keyspace: datastax
[default@datastax] _
```

Figure 6-14. Selecting the datastax keyspace

16. To output the table stored in Cassandra run the following commands in Cassandra-CLI.

```
assume catalog keys as utf8;
assume catalog validator as utf8;
assume catalog comparator as utf8;
GET catalog[utf8('catalog1')];
GET catalog[utf8('catalog2')];
```

The two rows stored in the catalog table get listed as shown in Figure 6-15.

```
[default@unknown] use datastax;
Authenticated to keyspace: datastax
[default@datastax] assume catalog keys as utf8;
Assumption for column family 'catalog' added successfully.
[default@datastax] assume catalog validator as utf8;
Assumption for column family 'catalog' added successfully.
[default@datastax] assume catalog comparator as utf8;
Assumption for column family 'catalog' added successfully.
[default@datastax] GET catalog[utf8<'catalog1'>];
=> <name= **Oracle Magazine , value=, timestamp=1439246341111000>
=> <name= **Oracle Magazine *author , value=David A. Kelly, timestamp=1439246341111000>
=> <name= **Oracle Magazine edition , value=November–December 2013, timestamp=1439246341111000>
=> <name= **Oracle Magazine publisher , value=Oracle Publishing, timestamp=1439246341111000>
=> <name= **Oracle Magazine *title , value=Engineering as a Service, timestamp=1439246341111000>
Returned 5 results.
Elapsed time: 114 msec(s).
[default@datastax] GET catalog[utf8<'catalog2'>];
=> <name= **Oracle Magazine , value=, timestamp=1439246341359000>
=> <name= **Oracle Magazine *author , value=Tom Haunert, timestamp=1439246341359000>
=> <name= **Oracle Magazine edition , value=November–December 2013, timestamp=1439246341359000>
=> <name= **Oracle Magazine publisher , value=Oracle Publishing, timestamp=1439246341359000>
=> <name= **Oracle Magazine *title , value=Quintessential and Collaborative, timestamp=1439246341359000>
Returned 5 results.
Elapsed time: 16 msec(s).
[default@datastax] _
```

Figure 6-15. Listing the Catalog Table Rows

Next, we shall migrate the Cassandra data to MongoDB server.

Migrating the Cassandra Table to MongoDB

In this section we shall get the data stored in Cassandra database and migrate the data to MongoDB server. We shall use the `MigrateCassandraToMongoDB` class to migrate the data from Cassandra database to MongoDB server.

1. Add a method called `migrate()` to the `MigrateCassandraToMongoDB` class and invoke the method from the `main` method.
2. From the `MigrateCassandraToMongoDB` class connect to the Cassandra server as explained in the previous section in the `main` method.

```
cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
session = cluster.connect();
```

A `Session` object is created to represent a connection with Cassandra server. We shall use the `Session` object to run a `SELECT` statement on Cassandra to select the data to be migrated.

3. Run a SELECT statement as follows to select all rows from the catalog table in the datastax keyspace in the migrate() method.

```
ResultSet results = session.execute("select * from datastax.catalog");
```

4. The result set of the query is represented with the ResultSet class. A row in the ResultSet is represented with the Row class. Iterate over the ResultSet to fetch each row as a Row object.

```
for (Row row : results) {  
}
```

Before we migrate the rows of data fetched from Cassandra, create a Java client for MongoDB because we would need to connect to MongoDB and add the fetched data to MongoDB.

5. The MongoClient class represents a client to MongoDB and provides internal connection pooling. We shall use the MongoClient(List<ServerAddress> seeds) constructor with the ServerAddress instance constructed from the host as localhost and the port on which the server is running as 27017. In the migrate() method create a MongoClient instance.

```
MongoClient mongoClient = new MongoClient(Arrays.asList(new ServerAddress(  
"localhost", 27017)));
```

6. A database instance is represented with the com.mongodb.client.MongoDatabase class. Create a database object for the local database.

```
MongoDatabase db = mongoClient.getDatabase("local");
```

7. A MongoDB database collection is represented with the com.mongodb.client.MongoCollection class. Next, create a MongoDB collection instance using the getCollection(String collectionName) method of the MongoDatabase object. Create a collection called catalog.

```
MongoCollection<Document> coll = db.getCollection("catalog");
```

8. A MongoDB collection gets created implicitly when a collection is referenced by name without having to first create the collection. Next, we shall migrate the result set obtained from Cassandra to MongoDB using a for loop to iterate over the rows in the result set.

```
for (Row row : results) {  
}
```

9. The ColumnDefinitions class represents the metadata describing the columns contained in a ResultSet. Obtain the column definitions as represented by a ColumnDefinitions object using the getColumnDefinitions() method of Row. Create an Iterator over the ColumnDefinitions object using the iterator() method with each column definition being represented with ColumnDefinitions.Definition.

```
ColumnDefinitions columnDefinitions = row.getColumnDefinitions();  
Iterator<ColumnDefinitions.Definition> iter = columnDefinitions.iterator();
```

10. Using a while loop and the hasNext method of Iterator iterate over the columns and obtain the column names using the getName method in ColumnDefinitions. Definition.

```
while (iter.hasNext()) {
    ColumnDefinitions.Definition column = iter.next();
    String columnName = column.getName();
}
```

11. Within the while loop, using the getString(String columnName) method in Row obtain the value corresponding to each column.

```
String columnValue = row.getString(columnName);
```

The org.bson.Document class represents a MongoDB document as a Map; key/value map that may be stored in Mongo database. The org.bson.Document implements the org.bson.conversions.Bson interface and represents a basic BSON object stored in MongoDB server.

12. Next, we shall add a BSON document to the Mongo database. Within the for loop create a org.bson.Document instance using the class constructor.

```
Document catalog = new Document();
```

13. Once a org.bson.Document has been created key/value pairs may be added using the append(String key, Object value) method in org.bson.Document class. Use the column name/value pairs obtained from Cassandra result set to create a complete BSON document using the append method.

```
catalog = catalog.append(columnName, columnValue);
```

14. As discussed in Chapter 1 the MongoCollection class provides the insertOne(TDocument document) method to add a single document. Save the Document instance using the insertOne(TDocument document) method.

```
coll.insertOne(catalog);
```

15. Next, output the document added to the MongoDB collection that also verifies that the document did get added. All the documents in a collection may be fetched using the find() method in MongoCollection. The find() method returns as result a FindIterable object.

```
FindIterable<Document> iterable = coll.find();
```

16. As discussed in Chapter 1, output the key/value pairs for each document stored in the FindIterable object. Use an enhanced for loop to obtain the Document instances in the FindIterable. Obtain the key set associated with each Document instance using the keySet() method, which returns a Set<String> object. Create an Iterator<String> object from the Set object using iterator(). Use a while loop to iterate over the key set and output the document key for each Document and the associated Document object.

```

FindIterable<Document> iterable = coll.find();
    String documentKey = null;
    for (Document document : iterable) {
        Set<String> keySet = document.keySet();
        Iterator<String> iter = keySet.iterator();
        while (iter.hasNext()) {
            documentKey = iter.next();
            System.out.println(documentKey);
            System.out.println(document.get(documentKey));
        }
    }
}

```

17. Close the MongoClient object using the `close()` method.

```
mongoClient.close();
```

18. Also shut down the Cassandra session and cluster.

```
session.close();
cluster.close();
```

The `MigrateCassandraToMongoDB` class is listed:

```

package mongodb;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.ColumnDefinitions;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

public class MigrateCassandraToMongoDB {
    private static Cluster cluster;
    private static Session session;
    private static MongoClient mongoClient;
    public static void main(String[] args) {
        cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
        session = cluster.connect();
        session = cluster.connect();
        migrate();
        mongoClient.close();
        session.close();
        cluster.close();
    }
}

```

```
public static void migrate() {
    mongoClient = new MongoClient(Arrays.asList(new ServerAddress(
        "localhost", 27017)));
    MongoDatabase db = mongoClient.getDatabase("local");
    MongoCollection<Document> coll = db.getCollection("catalog");
    ResultSet results = session.execute("select * from datastax.catalog");
    for (Row row : results) {
        ColumnDefinitions columnDefinitions = row.getColumnDefinitions();
        Iterator<ColumnDefinitions.Definition> iter = columnDefinitions
            .iterator();
        Document catalog = new Document();
        while (iter.hasNext()) {
            ColumnDefinitions.Definition column = iter.next();
            String columnName = column.getName();
            String columnValue = row.getString(columnName);
            catalog = catalog.append(columnName, columnValue);
        }
        coll.insertOne(catalog);
    }
    FindIterable<Document> iterable = coll.find();
    String documentKey = null;
    for (Document document : iterable) {
        Set<String> keySet = document.keySet();
        Iterator<String> iter = keySet.iterator();
        while (iter.hasNext()) {
            documentKey = iter.next();
            System.out.println(documentKey);
            System.out.println(document.get(documentKey));
        }
    }
}
```

19. To migrate the Cassandra table to MongoDB, right-click on `MigrateCassandraToMongoDB` class in Package Explorer and select Run As ➤ Java Application as shown in Figure 6-16.

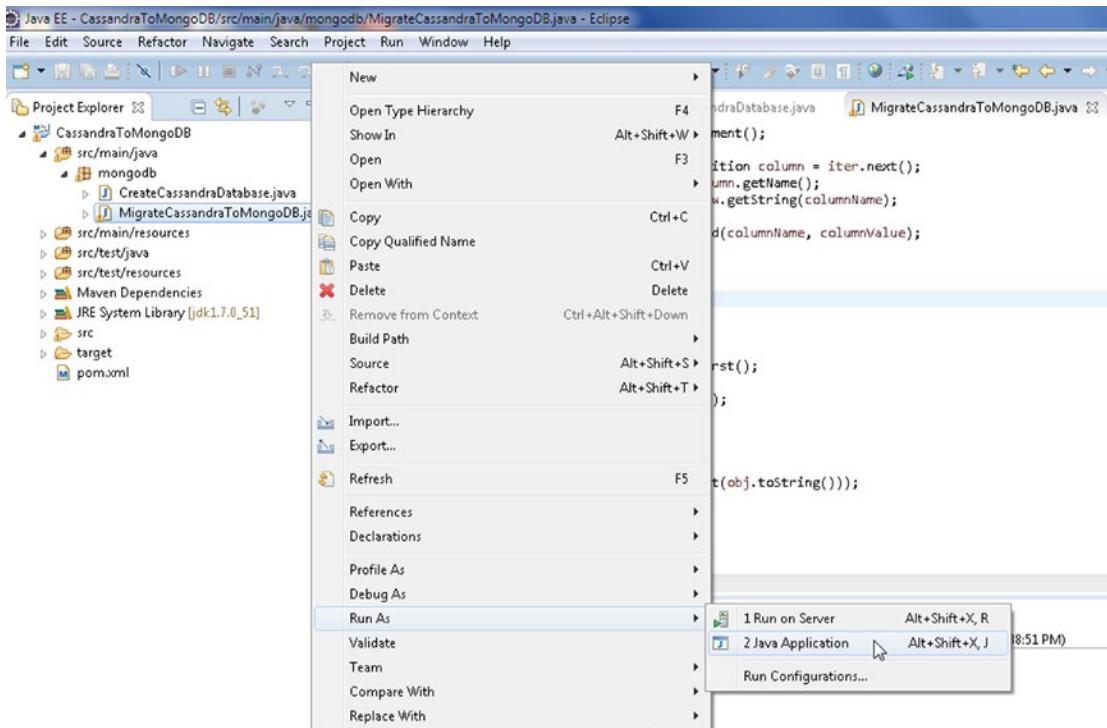


Figure 6-16. Running *MigrateCassandraToMongoDB.java* Application

The Apache Cassandra table gets migrated to MongoDB. The table data migrated to MongoDB is also output in the Eclipse console as shown in Figure 6-17.



The screenshot shows a Java application window titled "Console". The title bar also includes the text "<terminated> MigrateCassandraToMongoDB [Java Application]". The main content area displays the output of the migration process, listing two documents. Each document is represented by a series of key-value pairs:

- Document 1:**
 - _id: 55c93465cd73501ae89a80ba
 - catalog_id: catalog1
 - catalog1: Oracle Magazine
 - journal: Oracle Publishing
 - author: David A. Kelly
 - edition: November-December 2013
 - publisher: Oracle Publishing
 - title: Engineering as a Service
 - _id: 55c93465cd73501ae89a80bb
 - catalog_id: catalog2
 - catalog2: Oracle Magazine
 - journal: Tom Haunert
 - author: Tom Haunert
 - edition: November-December 2013
 - publisher: Oracle Publishing
 - title: Quintessential and Collaborative

Figure 6-17. Outputting the Migrated Documents

A more detailed output from the application is as shown below.

```
16:31:48.058 [main] INFO com.datastax.driver.core.Cluster - New Cassandra host /127.0.0.1:9042 added
16:31:48.143 [cluster1-nio-worker-1] DEBUG com.datastax.driver.core.Connection - Connection[/127.0.0.1:9042-2, inflight=0, closed=false] Connection opened successfully
16:31:48.168 [cluster1-nio-worker-1] DEBUG com.datastax.driver.core.Session - Added connection pool for /127.0.0.1:9042
16:31:48.176 [cluster1-nio-worker-2] DEBUG com.datastax.driver.core.Connection - Connection[/127.0.0.1:9042-3, inflight=0, closed=false] Connection opened successfully
16:31:48.190 [cluster1-nio-worker-2] DEBUG com.datastax.driver.core.Session - Added connection pool for /127.0.0.1:9042
16:31:49.122 [main] INFO org.mongodb.driver.cluster - Cluster created with settings {hosts=[localhost:27017], mode=MULTIPLE, requiredClusterType=UNKNOWN, serverSelectionTimeout='30000 ms', maxWaitQueueSize=500}
16:31:49.122 [main] INFO org.mongodb.driver.cluster - Adding discovered server localhost:27017 to client view of cluster
16:31:49.165 [main] DEBUG org.mongodb.driver.cluster - Updating cluster description to {type=UNKNOWN, servers=[{address=localhost:27017, type=UNKNOWN, state=CONNECTING}]}
```

```

16:31:49.253 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:27017] INFO org.mongodb.driver.connection - Opened connection [connectionId{localValue:1, serverValue:16}] to localhost:27017
16:31:49.254 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:27017] DEBUG org.mongodb.driver.cluster - Checking status of localhost:27017
16:31:49.256 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:27017] INFO org.mongodb.driver.cluster - Monitor thread successfully connected to server with descripti on ServerDescription{address=localhost:27017, type=STANDALONE, state=CONNECTED, ok=true, version=Ser verVersion{versionList=[3, 0, 5]}, minWireVersion=0, maxWireVersion=3, electionId=null, maxDocumentS ize=16777216, roundTripTimeNanos=2118747}
16:31:49.258 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:27017] INFO org.mongodb.driver.cluster - Discovered cluster type of STANDALONE
16:31:49.259 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:27017] DEBUG org.mongodb.driver.cluster - Updating cluster description to {type=STANDALONE, servers=[{address=localhost:27017, type=STANDALONE, roundTripTime=2.1 ms, state=CONNECTED}]
16:31:49.281 [main] INFO org.mongodb.driver.connection - Opened connection [connectionId{localValue:2, serverValue:17}] to localhost:27017
16:31:49.307 [main] DEBUG org.mongodb.driver.protocol.insert - Inserting 1 documents into namespace local.catalog on connection [connectionId{localValue:2, serverValue:17}] to server localhost:27017
16:31:49.318 [main] DEBUG org.mongodb.driver.protocol.insert - Insert completed
16:31:49.319 [main] DEBUG org.mongodb.driver.protocol.insert - Inserting 1 documents into namespace local.catalog on connection [connectionId{localValue:2, serverValue:17}] to server localhost:27017
16:31:49.320 [main] DEBUG org.mongodb.driver.protocol.insert - Insert completed
16:31:49.354 [main] DEBUG org.mongodb.driver.protocol.query - Sending query of namespace local.catal og on connection [connectionId{localValue:2, serverValue:17}] to server localhost:27017
16:31:49.358 [main] DEBUG org.mongodb.driver.protocol.query - Query completed_id
55c93465cd73501ae89a80ba
catalog_id
catalog1
journal
Oracle Magazine
author
David A. Kelly
edition
November-December 2013
publisher
Oracle Publishing
title
Engineering as a Service
_id
55c93465cd73501ae89a80bb
catalog_id
catalog2
journal
Oracle Magazine
author
Tom Haunert

```

```

edition
November-December 2013
publisher
Oracle Publishing
title
Quintessential and Collaborative
16:31:49.364 [main] INFO org.mongodb.driver.connection - Closed connection
[connectionId{localValue:2, serverValue:17}] to localhost:27017 because the pool has been
closed.
16:31:49.365 [main] DEBUG org.mongodb.driver.connection - Closing connection
connectionId{localValue:2, serverValue:17}
16:31:49.368 [main] DEBUG com.datastax.driver.core.Connection -
Connection[/127.0.0.1:9042-3, inFlight ht=0, closed=true] closing connection
16:31:49.371 [cluster-ClusterId{value='55c93465cd73501ae89a80b9', description='null'}-localhost:2701 7] DEBUG org.mongodb.driver.connection - Closing connection
connectionId{localValue:1, serverValue:1 6}
16:31:49.381 [main] DEBUG com.datastax.driver.core.Cluster - Shutting down
16:31:49.382 [main] DEBUG com.datastax.driver.core.Connection -
Connection[/127.0.0.1:9042-1, inFlight ht=0, closed=true] closing connection
16:31:49.382 [main] DEBUG com.datastax.driver.core.Connection -
Connection[/127.0.0.1:9042-2, inFlight ht=0, closed=true] closing connection

```

20. Run the following commands in mongo shell.

```

>use local
>show collections
>db.catalog.find()

```

The two documents migrated to MongoDB get listed as shown in Figure 6-18.

```

Administrator: C:\Windows\system32\cmd.exe - mongo

C:\Users\Deepak Vohra>mongo
2015-08-10T20:12:14.986-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
> use local
switched to db local
> show collections
catalog
startup_log
system.indexes
>db.catalog.find()
{
  "_id" : ObjectId("55c93465cd73501ae89a80ba"),
  "catalog_id" : "catalog1",
  "journal" : "Oracle Magazine",
  "author" : "David A. Kelly",
  "edition" : "November-De
cember 2013",
  "publisher" : "Oracle Publishing",
  "title" : "Engineering as a Ser
vice"
}
{
  "_id" : ObjectId("55c93465cd73501ae89a80bb"),
  "catalog_id" : "catalog2",
  "journal" : "Oracle Magazine",
  "author" : "Tom Haunert",
  "edition" : "November-Decemb
er 2013",
  "publisher" : "Oracle Publishing",
  "title" : "Quintessential and Colla
borative"
}
>

```

Figure 6-18. Listing the Migrated Documents

Summary

In this chapter we migrated an Apache Cassandra table to MongoDB server using a Java application in Eclipse IDE. First, we added documents to Cassandra using the Cassandra Java driver. Subsequently we migrated the Cassandra documents to MongoDB. In the next chapter we shall migrate Couchbase database documents to MongoDB.

CHAPTER 7



Migrating Couchbase to MongoDB

Both MongoDB and Couchbase are document-centric NoSQL databases. Both are based on the same data storage model, which is JSON, with a slight difference that MongoDB is based on the BSON (binary JSON) data model, which provides a wider selection of data types. The JSON support in Couchbase is relatively less developed. MongoDB stores documents in collections, which make it easier to handle documents. Mongo's support for JavaScript methods that run in the Mongo shell to perform CRUD operations on the documents is another advantage over Couchbase. To make use of these added features in MongoDB, it may be advantageous to migrate documents from Couchbase to MongoDB. In this chapter we shall migrate Couchbase documents to MongoDB. This chapter covers the following topics:

- Setting up the environment
- Creating a Maven project
- Creating Java classes
- Configuring the Maven project
- Adding a document to Couchbase
- Creating a Couchbase view
- Migrating a Couchbase document to MongoDB

Setting Up the Environment

We need to download the following software for this chapter.

- Couchbase Server Community or Enterprise Edition 3.0.x (or later version) couchbase-server-enterprise_3.0.3-windows_amd64.exe file from www.couchbase.com/nosql-databases/downloads. Double-click on the exe file to launch the installer and install Couchbase Server.
- Eclipse IDE for Java EE Developers from www.eclipse.org/downloads/.
- MongoDB 3.05 (or a later version) Windows binaries mongodb-win32-x86_64-3.0.5-signed.msi from www.mongodb.org/downloads. Double-click on the mongodb-win32-x86_64-3.0.5-signed file to install MongoDB 3.05. Add the bin directory, for example C:\Program Files\MongoDB\Server\3.0\bin, to the PATH environment variable.
- Java 7 from www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Create a directory C:\data\db for the MongoDB data if not already created for an earlier chapter. Start MongoDB with the following command from a command shell.

```
>mongod
```

MongoDB gets started waiting for connections on port 27017.

Log in to the Couchbase Console. Click on Data Buckets in the Couchbase Admin Console, which is accessed with URL localhost:8091. The default bucket should be listed in Couchbase Buckets. Click on the Documents button for the default bucket. Initially the default bucket should not have any documents in it as shown in Figure 7-1.

A screenshot of the Couchbase Admin Console interface. At the top, there's a navigation bar with links for Documentation, Support Forums, About, and Sign Out. Below the navigation bar, there's a main menu with Cluster Overview, Server Nodes, Data Buckets, Views, XDCR, Log, and Settings. The 'Data Buckets' option is selected. In the center, there's a search bar with 'default' and a dropdown arrow, followed by a breadcrumb trail showing 'default > Documents'. To the right of the search bar, there are buttons for Current page: 1, a page number input field containing '5', and a dropdown arrow. Below the search bar, there's a 'Documents Filter' section with a dropdown arrow, and buttons for Document ID, Lookup Id, and Create Document. There are two columns: 'ID' and 'Content'. A message at the bottom of the table area says 'There are currently no documents in this bucket.'.

Figure 7-1. Empty Couchbase Bucket default

Creating a Maven Project

We shall use a Maven project to create Couchbase documents and subsequently migrate the Couchbase documents to MongoDB. Next, create a Maven project in Eclipse.

1. Select File ▶ New ▶ Other.
2. In the New window, select Maven ▶ Maven Project and click on Next as shown in Figure 7-2.

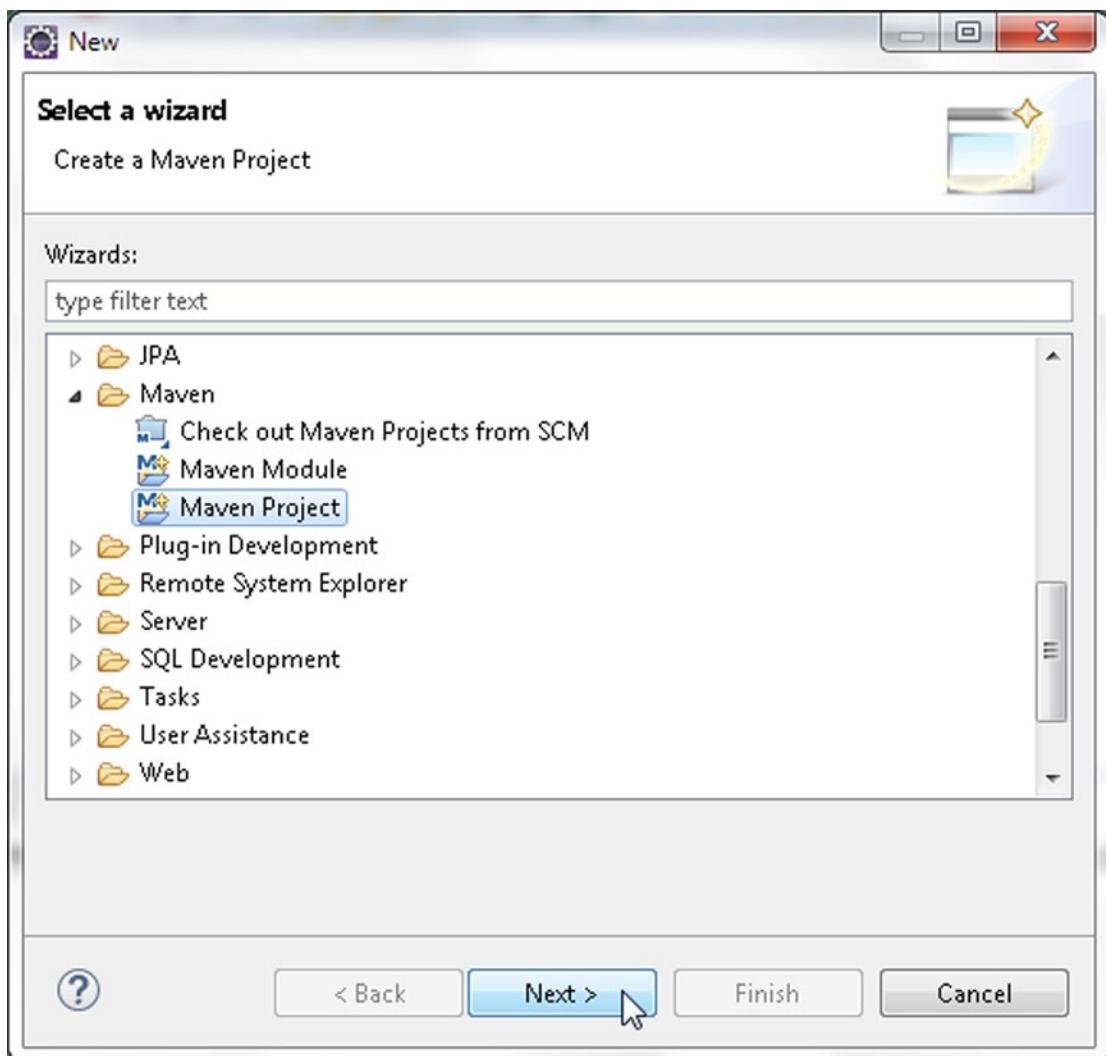


Figure 7-2. Selecting Maven ► Maven Project

3. In the New Maven Project wizard select the “Create a simple project” check box and the “Use default Workspace location” check box, and click on Next as shown in Figure 7-3.

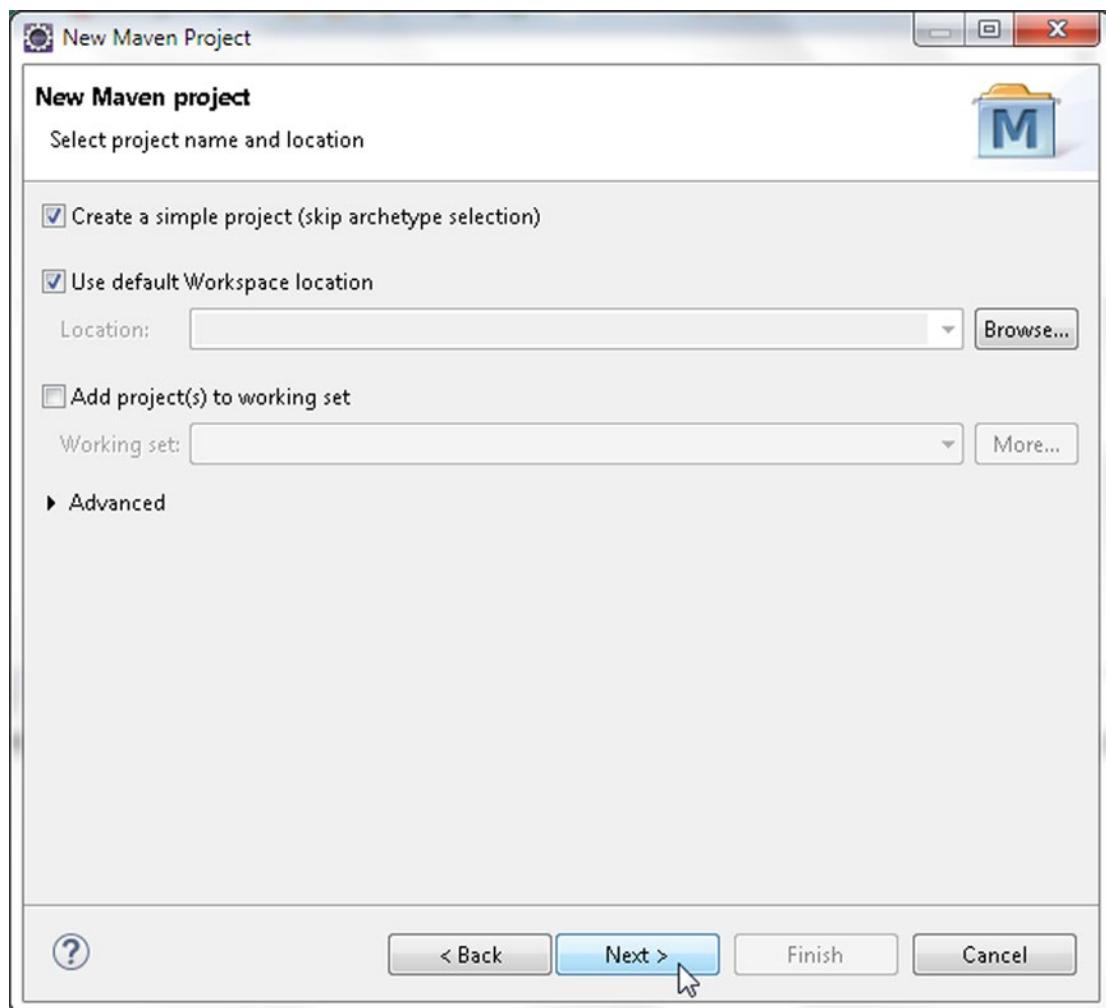


Figure 7-3. Creating a New Maven Project

4. To create the Maven project, specify the following and click on Finish as shown in Figure 7-4.
 - Group Id: com.mongodb.migration
 - Artifact Id: CouchbaseToMongoDB
 - Version: 1.0.0
 - Packaging: jar
 - Name: CouchbaseToMongoDB

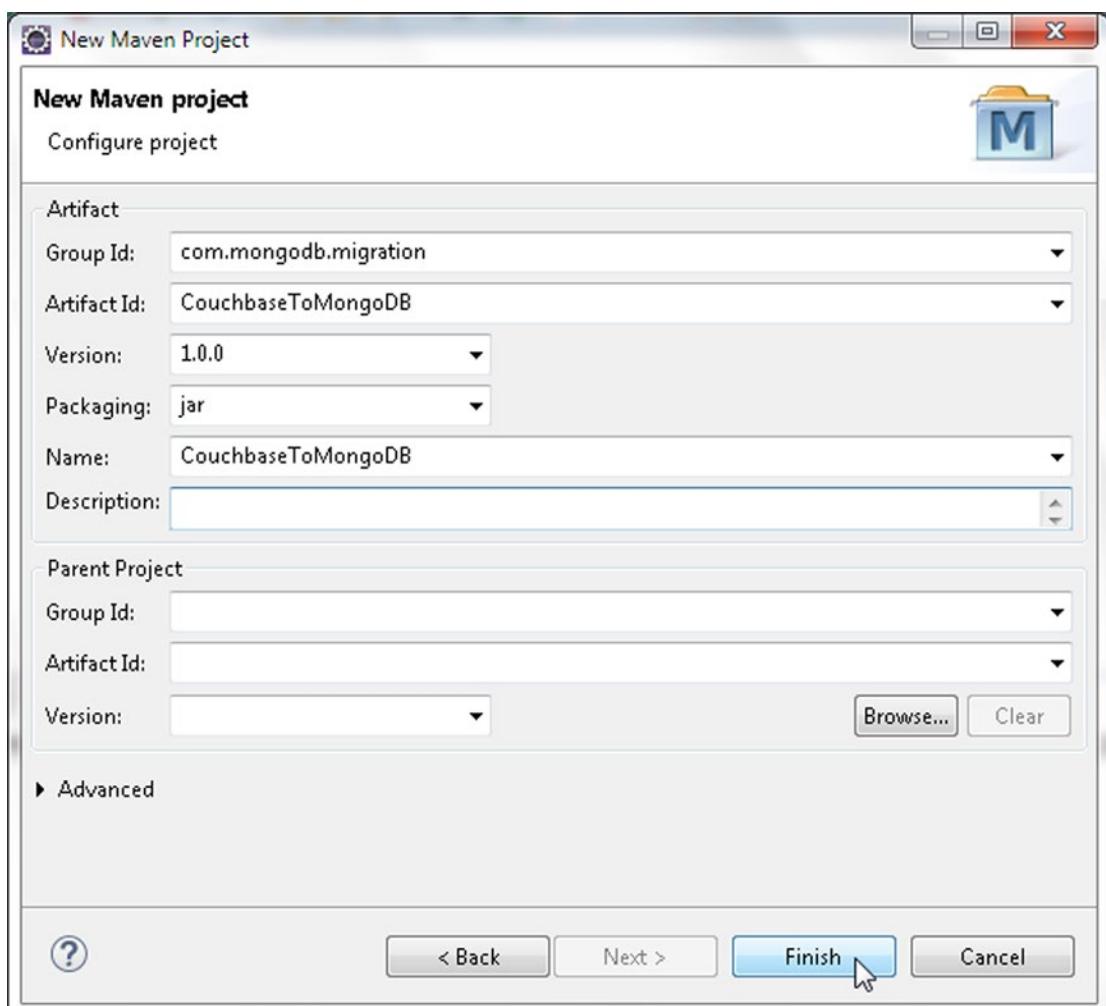


Figure 7-4. Specifying Maven Project Artifacts

A Maven project gets added to the Package Explorer in Eclipse as shown in Figure 7-5.

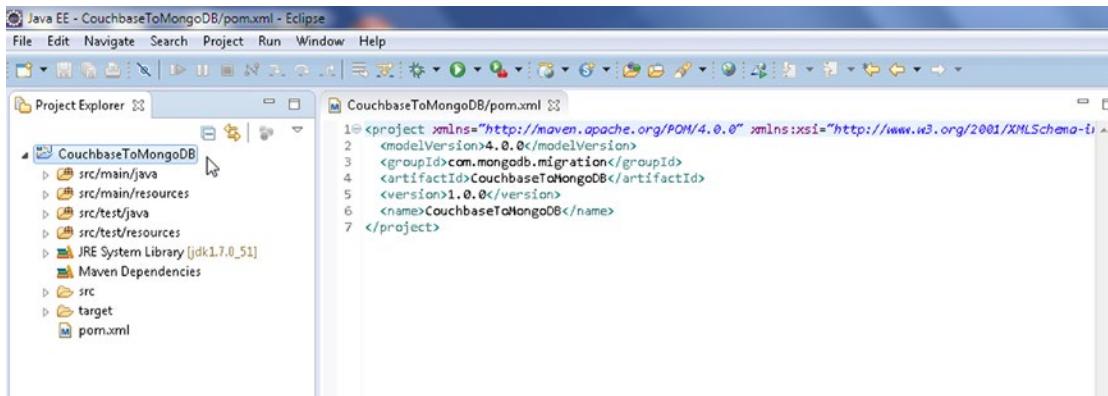


Figure 7-5. Maven Project in Package Explorer

Creating Java Classes

We shall migrate a MongoDB database document to Couchbase Server in a Java application. Create two classes: `CreateCouchbaseDocument` and `MigrateCouchbaseToMongoDB`.

1. To create a Java class select File ▶ New ▶ Other.
2. In the New window, select Java ▶ Class and click on Next as shown in Figure 7-6.

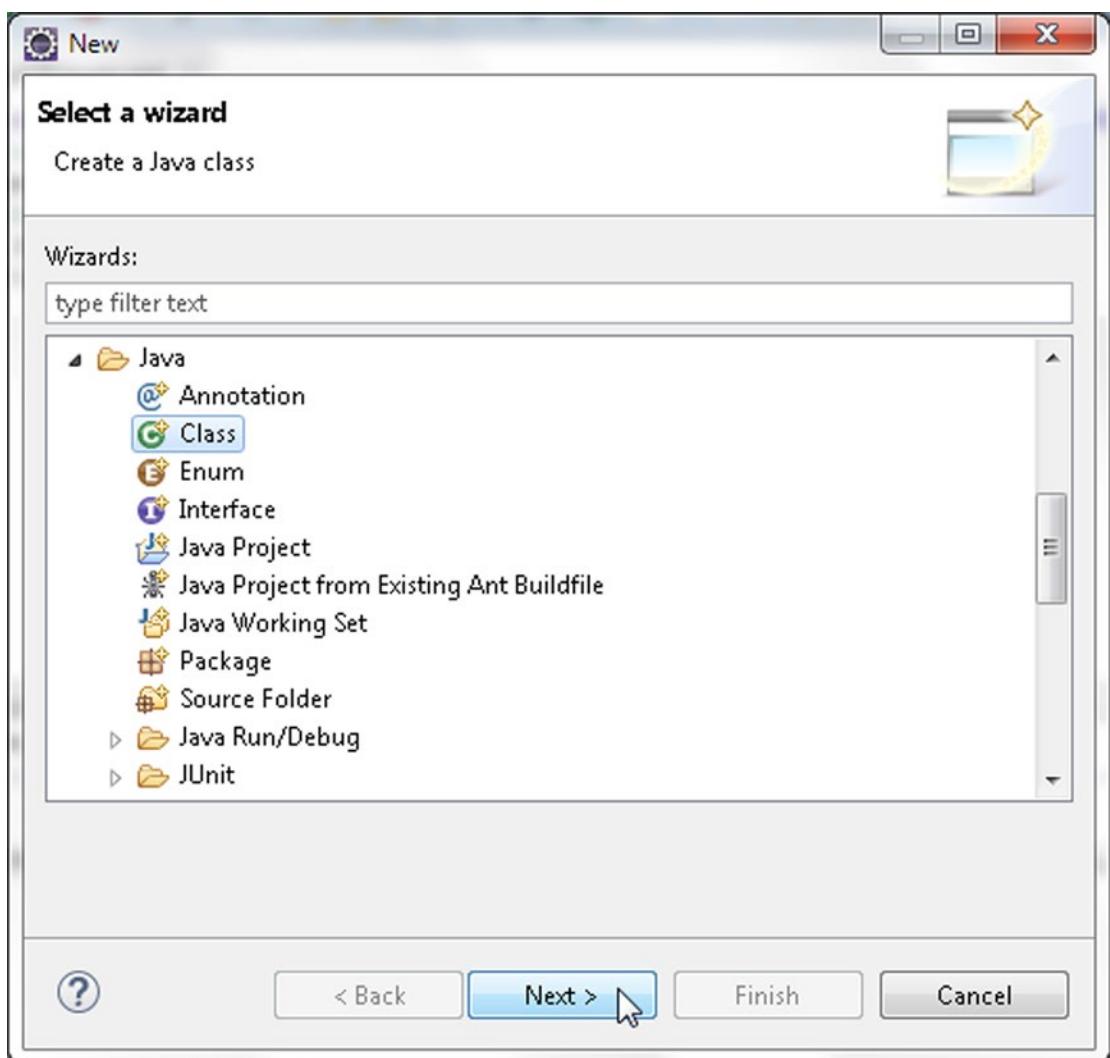
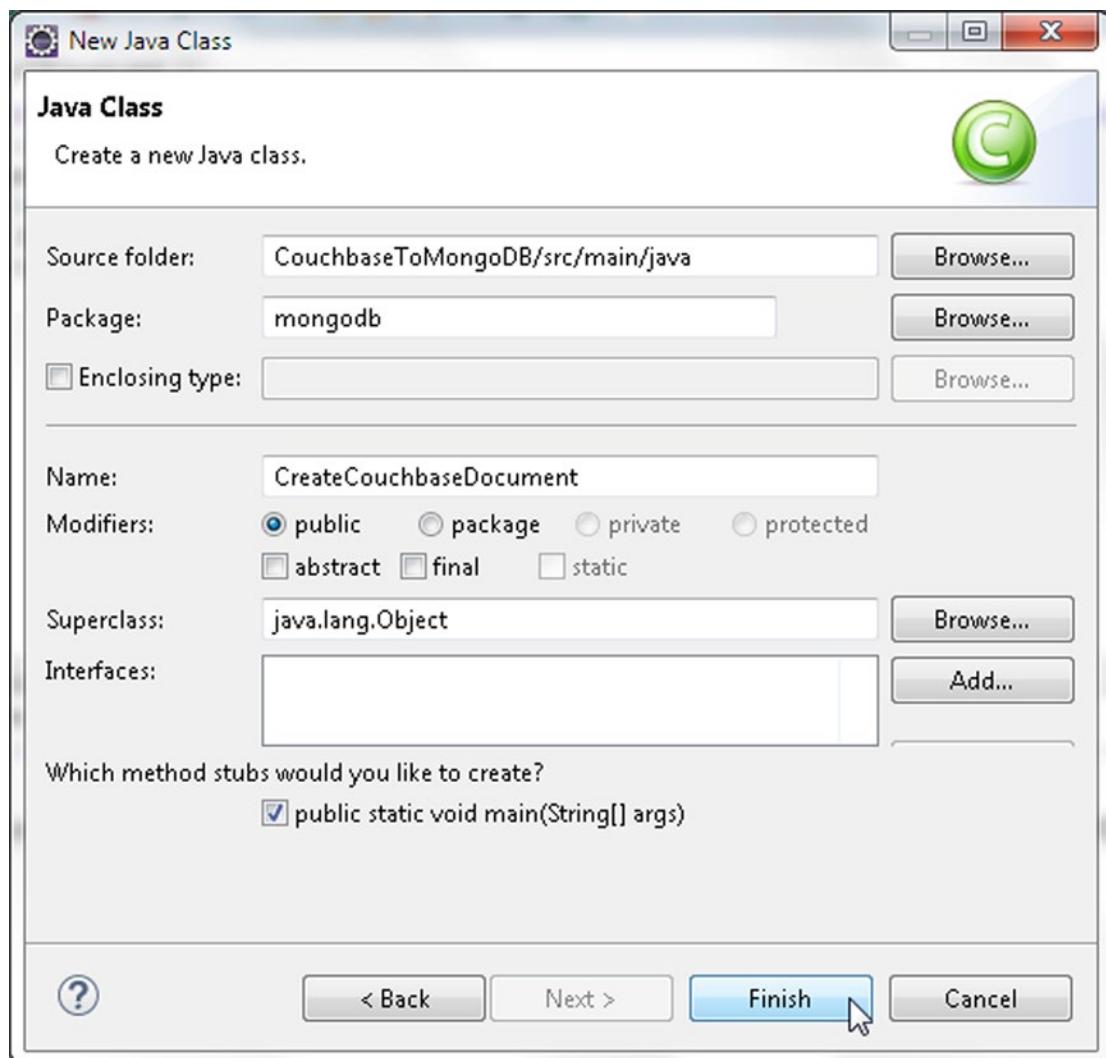


Figure 7-6. Selecting Java ► Java Class

3. In New Java Class wizard select the Source folder and specify Package as `mongodb`. Specify class Name as `CreateCouchbaseDocument` and click on Finish as shown in Figure 7-7.

**Figure 7-7.** New Java Class Wizard

4. Similarly, add a class `MigrateCouchbaseToMongoDB` as shown in Figure 7-8. The two classes `CreateMongoDB` and `MigrateMongoDBToCouchbase` are shown in Package Explorer as shown in Figure 7-8.

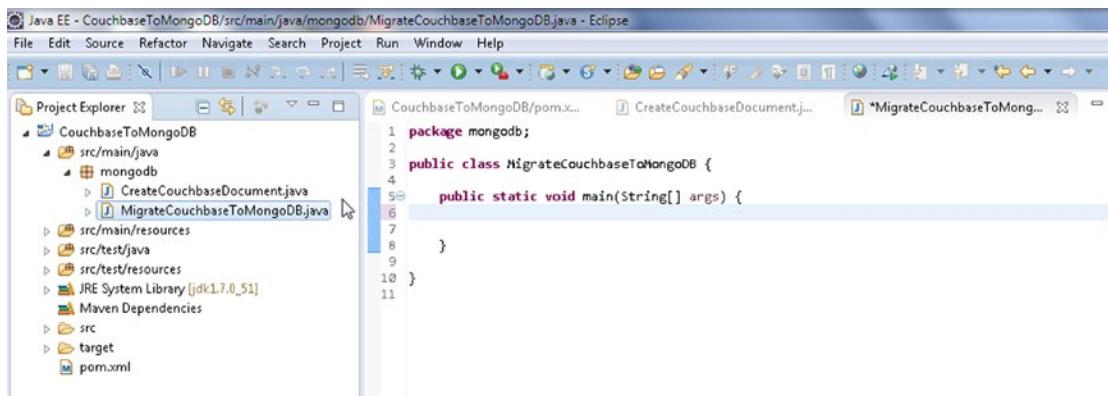


Figure 7-8. Java Classes in Package Explorer

Configuring the Maven Project

We need to add some Maven dependencies to the project classpath. Add the dependencies listed in Table 7-1 to `pom.xml` configuration file in the Maven project.

Table 7-1. Maven Dependencies

Dependency	Description
Mongo Java Driver 3.0.3	The MongoDB Java driver required to access MongoDB from a Java application.
Couchbase Server Java SDK Client library 2.1.4	The Java Client to Couchbase Server.
Apache Commons BeanUtils 1.9.2	Utility Jar for Java classes developed with the JavaBeans pattern.
Apache Commons Collections 3.2.1	Java Collections framework provides data structures that accelerate development.
Apache Commons Logging 1.2	An interface for common logging implementations.

The `pom.xml` is listed below.

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mongodb.migration</groupId>
  <artifactId>CouchbaseToMongoDB</artifactId>
  <version>1.0.0</version>
  <name>CouchbaseToMongoDB</name>

```

```
<dependencies>
    <dependency>
        <groupId>com.couchbase.client</groupId>
        <artifactId>java-client</artifactId>
        <version>2.1.4</version>
    </dependency>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>3.0.3</version>
    </dependency>
    <dependency>
        <groupId>commons-beanutils</groupId>
        <artifactId>commons-beanutils</artifactId>
        <version>1.9.2</version>
    </dependency>
    <dependency>
        <groupId>commons-collections</groupId>
        <artifactId>commons-collections</artifactId>
        <version>3.2.1</version>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
</project>
```

Select File ► Save All to save the `pom.xml` configuration file. The required jar files get downloaded, and get added to the Java build path. To find which Jars have been added to the Maven project Java build path, right-click on the project node in Package Explorer and select Properties. In Properties select Java Build Path. The Jars added to the migration project are shown in Figure 7-9.

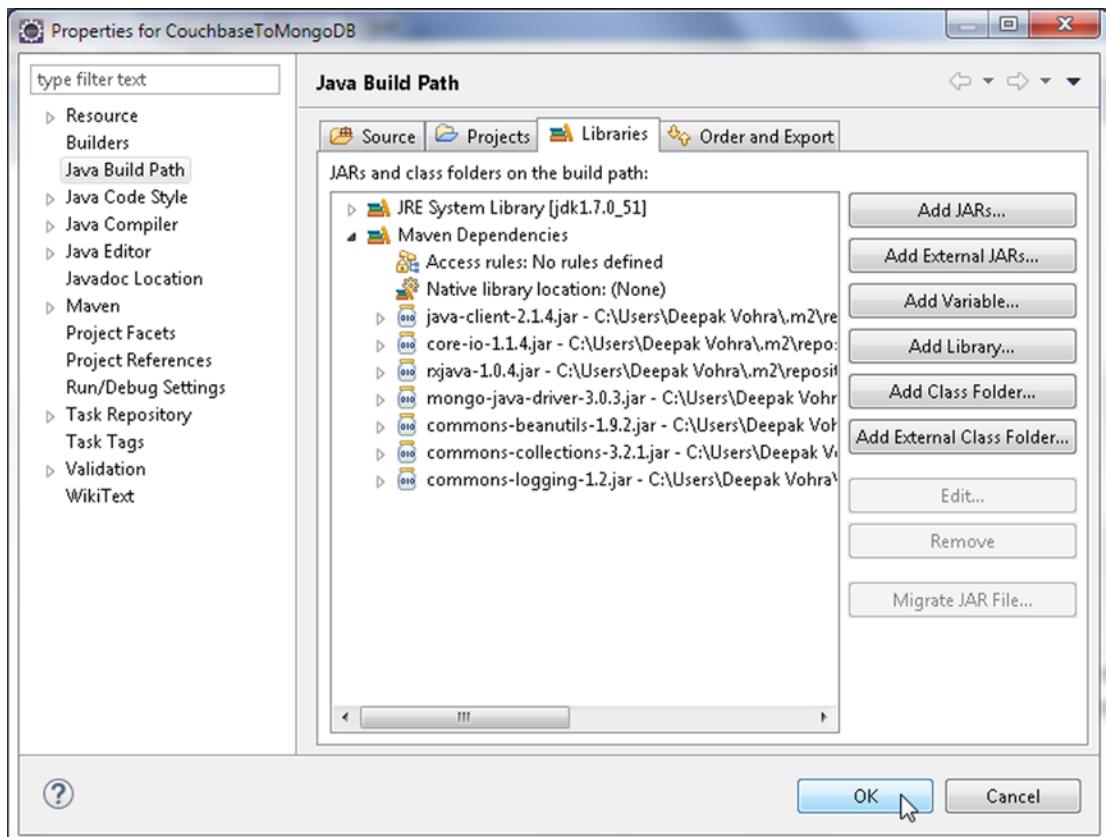


Figure 7-9. Jar Files in Java Build Path

Adding Documents to Couchbase

The `com.couchbase.client.java.CouchbaseCluster` class is the client class for Couchbase Server and is the entry point to access Couchbase cluster, which may consist of one or more servers. In the `CreateCouchbaseDocument` application we shall use the `CouchbaseCluster` class to create and store a JSON document in Couchbase Server. The `CouchbaseCluster` class provides the overloaded `create()` methods to create an instance of `CouchbaseCluster`. We shall use the static method `create()` that does not take any args and is used to connect to the default bucket at localhost at port 8091. Couchbase Server stores documents in Data Buckets. The Bucket interface represents a connection to a bucket to perform operations on the bucket synchronously.

1. Create a `CouchbaseCluster` instance and subsequently connect to the default bucket using the `openBucket()` method. For connecting to the “default” bucket, bucket name and password are not required to be specified.

```
Cluster cluster = CouchbaseCluster.create();
Bucket defaultBucket = cluster.openBucket();
```

The `com.couchbase.client.java.document.json.JsonObject` class represents a JSON object stored in Couchbase Server. A document is represented with the `com.couchbase.client.java.document.Document` interface and several class implementations are provided, including the `com.couchbase.client.java.document.JsonDocument`, which creates a document from a `com.couchbase.client.java.document.json.JsonObject`. `JsonObject` represents a JSON object, the `{a1:v1,a2:v2}` JSON stored in Couchbase. The `JsonObject` class provides static methods `empty()` and `create()` to create an empty `JsonObject` instance. The `JsonObject` class provides the overloaded `put` methods to put field/value pairs in a `JsonObject` instance. The field name in each of these methods is of type `String`. A `put` method is provided for each of the value types `String`, `int`, `long`, `double`, `boolean`, `JsonObject`, `JsonArray`, and `Object`. We shall make use of the `put(String, String)` method to add key/value pairs to a JSON document.

2. Create a `JsonObject` instance for a JSON document with fields `journal`, `publisher`, `edition`, `title`, and `author` with `String` values using the `put(java.lang.String name, java.lang.String value)` method. First, invoke the `empty()` method to return an empty `JsonObject` instance and subsequently invoke the `put(java.lang.String name, java.lang.String value)` method to add field/value pairs.

```
JsonObject catalogObj = JsonObject.empty()
    .put("journal", "Oracle Magazine")
    .put("publisher", "Oracle Publishing")
    .put("edition", "March April 2013")
    .put("title", "Engineering as a Service")
    .put("author", "David A. Kelly");
```

3. The `Bucket` class provides several overloaded `insert` and `upsert` methods to add a document to a bucket. Create an instance of `JsonDocument` using the `JsonDocument.create(java.lang.String id, JsonObject content)` method with document id as “catalog.” Use the `insert(D document)` class to add a `JsonObject` instance to the default bucket.

```
defaultBucket.insert(JsonDocument.create("catalog1", catalogObj));
```

4. Similarly, add another JSON document to the default bucket.

```
catalogObj = JsonObject.empty()
    .put("journal", "Oracle Magazine")
    .put("publisher", "Oracle Publishing")
    .put("edition", "March April 2013")
    .put("title", "Quintessential and Collaborative")
    .put("author", "Tom Haunert");
defaultBucket.insert(JsonDocument.create("catalog2", catalogObj));
```

5. After adding documents disconnect from the Couchbase cluster using the `disconnect()` method.

```
cluster.disconnect();
```

The CreateCouchbaseDocument class is listed below.

```
package mongodb;
import com.couchbase.client.java.Bucket;
import com.couchbase.client.java.Cluster;
import com.couchbase.client.java.CouchbaseCluster;
import com.couchbase.client.java.document.JsonDocument;
import com.couchbase.client.java.document.json.JsonObject;

public class CreateCouchbaseDocument {

    public static void main(String args[]) {
        Cluster cluster = CouchbaseCluster.create();
        Bucket defaultBucket = cluster.openBucket();
        JsonObject catalogObj = JsonObject.empty()
            .put("journal", "Oracle Magazine")
            .put("publisher", "Oracle Publishing")
            .put("edition", "March April 2013")
            .put("title", "Engineering as a Service")
            .put("author", "David A. Kelly");
        defaultBucket.insert(JsonDocument.create("catalog1", catalogObj));
        catalogObj = JsonObject.empty()
            .put("journal", "Oracle Magazine")
            .put("publisher", "Oracle Publishing")
            .put("edition", "March April 2013")
            .put("title", "Quintessential and Collaborative")
            .put("author", "Tom Haunert");
        defaultBucket.insert(JsonDocument.create("catalog2", catalogObj));
        cluster.disconnect();
    }
}
```

6. To run the `CreateCouchbaseDocument.java` application right-click on the class in Package Explorer and select Run As ▶ Java Application as shown in Figure 7-10.

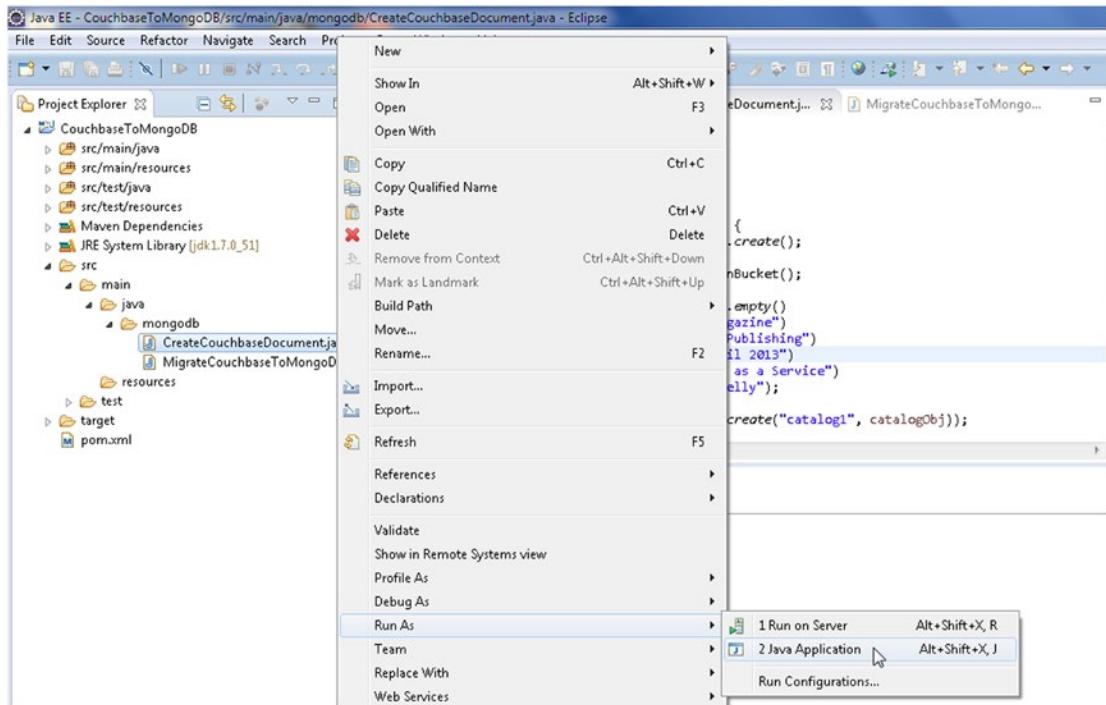


Figure 7-10. Running the *CreateCouchbaseDocument.java* Application

The JSON documents get stored in the Couchbase Server. Log in to the Couchbase Server Administration Console if not already logged in. Click on Data Buckets. The Item Count for the “default” bucket should be listed as 2 as shown in Figure 7-11. Click on Documents to list the documents added.

The screenshot shows the Couchbase Administration Console. The top navigation bar includes links for Documentation, Support Forums, About, and Sign Out. The main menu has tabs for Cluster Overview, Server Nodes, Data Buckets (which is currently selected), Views, XDCR, Log, and Settings. Below the menu, the 'Data Buckets' section is titled 'Couchbase Buckets'. It lists the 'default' bucket with the following details:

Bucket Name	Nodes	Item Count	Ops/sec	Disk Fetches/sec	RAM/Quota Usage	Data/Disk Usage
default	1	2	1.97	0	30.6MB / 4.11GB	90MB / 90.1MB

Below the table are two buttons: 'Documents' and 'Views'. The 'Documents' button is highlighted with a mouse cursor.

Figure 7-11. Item Count is 2

7. The two documents added get listed as shown in Figure 7-12. Click on Edit Document to display the JSON for a document.

ID	Content	
catalog1	{ "journal": "Oracle Magazine", "edition": "March April 2013..."}	Edit Document Delete
catalog2	{ "journal": "Oracle Magazine", "edition": "March April 2013..."}	Edit Document Delete

Figure 7-12. Listing Documents in default Bucket

The catalog1 ID JSON document gets displayed as shown in Figure 7-13.

```

1 {
2   "journal": "Oracle Magazine",
3   "edition": "March April 2013",
4   "author": "David A. Kelly",
5   "title": "Engineering as a Service",
6   "publisher": "Oracle Publishing"
7 }

```

Figure 7-13. Listing catalog1 Document JSON

8. Similarly list the catalog2 document as shown in Figure 7-14.

The screenshot shows the Couchbase web interface. At the top, there's a navigation bar with links for Documentation, Support Forums, About, and Sign Out. Below the navigation is a header with tabs for Cluster Overview, Server Nodes, Data Buckets, Views, XDCR, Log, and Settings. A dropdown menu shows 'default' selected, with a path indicator 'Documents'. The main content area displays a JSON document named 'catalog2'. The JSON code is:

```

1 {
2   "journal": "Oracle Magazine",
3   "edition": "March April 2013",
4   "author": "Tom Haunert",
5   "title": "Quintessential and Collaborative",
6   "publisher": "Oracle Publishing"
7 }

```

Below the JSON code are three buttons: Delete, Save As..., and Save.

Figure 7-14. Listing catalog2 Document JSON

Creating a Couchbase View

The JSON data stored in Couchbase Server can be indexed using a view, which creates an index on the data according to the defined format and structure. A view extracts the fields from the JSON document object in Couchbase Server and creates an index that can be queried. A view is a logical structure, and a map function maps the fields of the JSON document object stored in the Couchbase Server to a view.

Optionally a reduce function can also be applied to summarize (or average or sum) the data. In this section we create a view on the JSON document in the Couchbase Server. A map function has the following format.

```

function(doc, meta)
{
  emit(doc.name, [doc.field1, doc.field2]);
}

```

When the function is translated to a `map()` function, the `map()` function is supplied with two arguments for each document stored in a bucket: the `doc` arg and the `meta` arg. The `doc` arg is the document object stored in the Couchbase bucket, and its content type can be identified with the `meta.type` field. The `meta` arg is the metadata for the document object stored in the bucket. Every document in the data bucket is submitted to the `map()` function. Within the `map()` function any custom code can be specified. The `emit()` function is used to emit a row or a record of data from the `map()` function. The `emit()` function takes two arguments: a key and a value.

```
emit(key,value)
```

The emitted key is used for sorting and querying the document object fields mapped to the view. The key may have any format such as a string, a number, a compound structure such as an array, or a JSON object. The value is the data to be output in a row or record and it may have any format including a string, number, an array, or JSON. Specify the following function for the mapping from the Couchbase Server bucket to the view. The function first tests if the type of the document is JSON and subsequently emits records with each record key being the document name and each record value being the data stored in the fields of the document object.

Next, create a view in Couchbase Console.

1. Select Data Buckets > default bucket. Subsequently, select View. The Development View tab is selected by default.
2. Click on Create Development View to create a development view as shown in Figure 7-15.

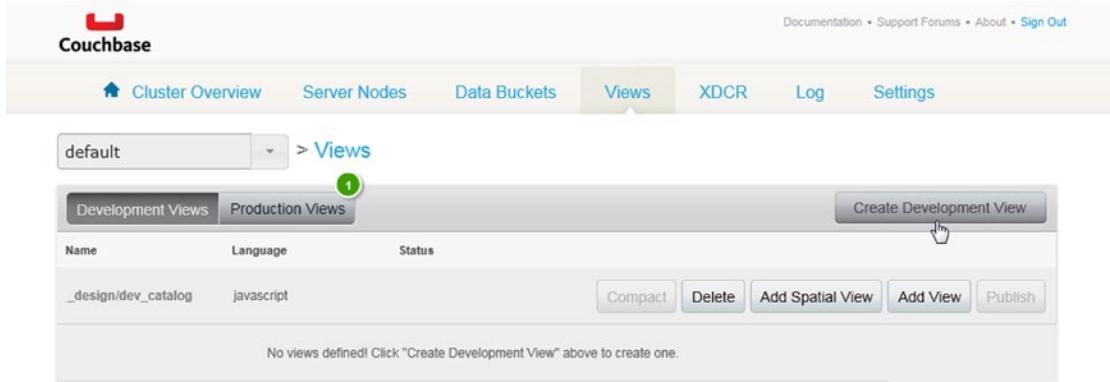


Figure 7-15. Selecting Create Development View

3. In Create Development View dialog specify a Design Document Name (_design/dev_catalog) and View Name (catalog_view) as shown in Figure 7-16. The _design prefix is not included in the design document name when accessed programmatically with a Java client. Click on Save.

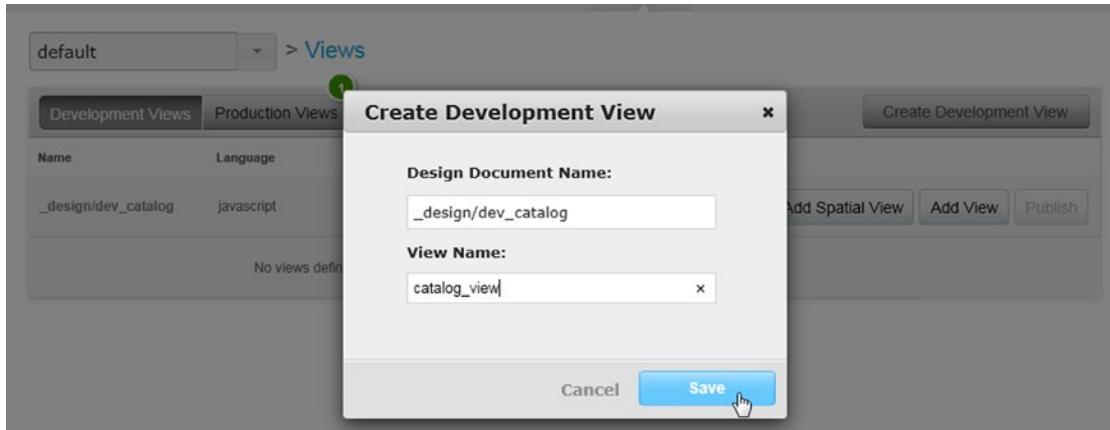


Figure 7-16. Creating a Development View

A development view called `catalog_view` gets created as shown in Figure 7-17.

The screenshot shows the Couchbase Web UI interface. At the top, there's a navigation bar with links for Cluster Overview, Server Nodes, Data Buckets, Views (which is the active tab), XDCR, Log, and Settings. Below the navigation bar, a dropdown menu shows 'default' selected. To the right of the dropdown is a breadcrumb trail: 'Views > Views'. Underneath the breadcrumb, there are two tabs: 'Development Views' (selected) and 'Production Views'. A green circular badge with the number '1' is positioned above the tabs. On the right side of the screen is a button labeled 'Create Development View'. The main content area displays a table with two rows. The first row contains the entry '_design/dev_catalog' with 'javascript' listed under 'Language'. To the right of this row are five buttons: Compact, Delete, Add Spatial View, Add View, and Publish. The second row contains the entry 'catalog_view'. To the right of this row are two buttons: Edit (with a hand cursor icon) and Delete. The entire interface has a clean, modern design with a light gray background and blue/gray UI elements.

Figure 7-17. The `catalog_view` View

4. We need to edit the default Map function to output a JSON with key/value pairs for the journal, publisher, edition, title, and author fields. Click on Edit to edit the view.
5. Copy the following function to the View Code ➤ Map region.

```
function(doc,meta) {  
    if (meta.type == 'json') {  
        emit(doc.name, [doc.journal,doc.publisher,doc.edition,doc.title,doc.author]);  
    } }  
}
```

6. Click on Save to save the Map function. The `catalog_view` view including the View Code is shown in Figure 7-18.

The screenshot shows the Couchbase Web UI interface. At the top, there's a navigation bar with links to Documentation, Support Forums, About, and Sign Out. Below the navigation bar, the main menu includes Cluster Overview, Server Nodes, Data Buckets, Views (which is currently selected), XDCR, Log, and Settings. The current view is under the 'Views' tab, specifically for the 'default' bucket and the '_design/dev_catalog/_view/catalog_view'. On the right side of the screen, there are two tabs: 'Preview a Random Document' and 'Edit Document'. The 'Edit Document' tab is active, showing the view code. Under the 'VIEW CODE' section, there are two parts: 'Map' and 'Reduce (built in: _count, _sum, _stats)'. The 'Map' section contains the following JavaScript code:

```

Map
1
function(doc,meta) {
2   if (meta.type == 'json') {
3     emit(doc.name, [doc.journal,doc.publisher,doc.edition,doc.title,doc.author]);
4   }
}

```

The 'Reduce' section is empty, containing only the number '1'. At the bottom of the code editor, there are 'Save As...' and 'Save' buttons. Below the code editor, there are 'Filter Results' and 'Show Results' buttons. A URL at the bottom of the page is: ?stale=false&connection_timeout=60000&limit=10&skip=0

Figure 7-18. Map Function of `catalog_view` View

7. We need to convert the development view to a production view before we are able to access the view from a Couchbase Java client. Click on Publish as shown in Figure 7-19 to convert the view to a production view.

The screenshot shows the Couchbase Web UI interface. At the top, there's a navigation bar with links to Documentation, Support Forums, About, and Sign Out. Below the navigation bar, the main menu includes Cluster Overview, Server Nodes, Data Buckets, Views (which is currently selected), XDCR, Log, and Settings. The current view is under the 'Views' tab, specifically for the 'default' bucket. There are two tabs: 'Development Views' (which is selected) and 'Production Views'. In the main area, there's a table with columns for Name, Language, and Status. One row in the table is for the view '_design/dev_catalog/_view/catalog_view', which is listed as 'javascript'. To the right of this row are buttons for Compact, Delete, Add Spatial View, Add View, and Publish. The 'Publish' button has a small circular icon with the number '1' above it, indicating that one or more changes have been made to the view. Below the table, there are 'Edit' and 'Delete' buttons.

Figure 7-19. Converting `catalog_view` to a Production View

Migrating Couchbase Documents to MongoDB

In this section we shall query the JSON documents stored earlier in Couchbase Server and migrate the JSON to MongoDB database. We shall use the `MigrateCouchbaseToMongoDB` application to migrate the JSON documents from Couchbase Server to a MongoDB database. We added a view encapsulated in a design document to Couchbase Server so that we may use the view to query the Couchbase Server. A view is represented with the `com.couchbase.client.java.view.View` class, and a view query is represented with the `com.couchbase.client.java.view.ViewQuery` class, which provides the `from(java.lang.String design, java.lang.String view)` class method to create a `ViewQuery` instance. The `Bucket` class provides the overloaded `query()` method to query a view. Each of the `query()` methods return a `ViewResult` instance, which represents the result from a `ViewQuery`. We shall generate a `ViewResult` for the documents stored in Couchbase Server using a view query and subsequently iterate over the view result to migrate the JSON documents to MongoDB.

1. In the `MigrateCouchbaseToMongoDB` application's `main` method create an instance of `Bucket` as discussed earlier.

```
Cluster cluster = CouchbaseCluster.create();
Bucket defaultBucket = cluster.openBucket();
```

2. Also as discussed in Chapter 1, create an instance of `MongoCollection` for the `catalog` collection to which the Couchbase documents are to be migrated.

```
mongoClient = new MongoClient(Arrays.asList(new ServerAddress("localhost", 27017)));
MongoDatabase db = mongoClient.getDatabase("local");
MongoCollection<Document> coll = db.getCollection("catalog");
```

3. Having created a connection with the Couchbase Server and the MongoDB server we shall migrate the Couchbase documents to MongoDB. Invoke the `query(ViewQuery query)` method using the `Bucket` instance to generate a `ViewResult` object. Create a `ViewQuery` argument using the static method `from(java.lang.String design, java.lang.String view)` with the design document name as `catalog` and view name as `catalog_view`, which were created in the preceding section.

```
ViewResult result = defaultBucket.query(ViewQuery.from("catalog", "catalog_view"));
```

4. `ViewResult` provides the overloaded `rows()` method that returns an `Iterator` over the rows in the view result. The `ViewRow` interface represents a view row. Using an enhanced for loop, iterate over the rows in the `ViewResult` and output each row to MongoDB.

```
for (ViewRow row : result) {
    //Migrate each row to MongoDB
}
```

5. A document in MongoDB Java driver is represented with the org.bson.Document class. Create an instance of Document for each row in the ViewResult. The JSON document in Couchbase Server driver is represented with the JsonDocument class. A JsonDocument instance may be obtained from a ViewRow instance using the document() method. Subsequently the JSON object is obtained from the JsonDocument with the content() method. The JsonObject instance has field/value pairs for a JSON document. Obtain the field names from the JsonObject as a Set using the getNames() method. Obtain an Iterator from the Set using the iterator() method. Using a while loop iterate over the field names and get each field name as a String. Obtain the field value using the getString(String fieldName) method in JsonObject. Using the append(String key, Object value) method in Document add the field/value pairs to the BSON document to be stored in MongoDB.

```

for (ViewRow viewRow : result) {
    Document catalog = new Document();
    JsonDocument json = viewRow.document();
    JsonObject jsonObj = json.content();
    Set<java.lang.String> fieldNames = jsonObj.getNames();
    Iterator<String> iter = fieldNames.iterator();
    while (iter.hasNext()) {
        String fieldName = iter.next();
        String fieldValue = jsonObj.getString(fieldName);
        catalog = catalog.append(fieldName, fieldValue);
    }
}

```

6. Having created the Document instance to be stored in MongoDB invoke the insertOne(TDocument document) method using the MongoCollection instance to store the BSON document in MongoDB.

```
coll.insertOne(catalog);
```

The MigrateCouchbaseToMongoDB application is listed below.

```

package mongodb;

import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
import org.bson.Document;
import com.couchbase.client.java.Bucket;
import com.couchbase.client.java.Cluster;
import com.couchbase.client.java.CouchbaseCluster;
import com.couchbase.client.java.document.JsonDocument;
import com.couchbase.client.java.document.json.JsonObject;
import com.couchbase.client.java.view.ViewQuery;
import com.couchbase.client.java.view.ViewResult;
import com.couchbase.client.java.view.ViewRow;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

```

```

public class MigrateCouchbaseToMongoDB {
    private static Bucket defaultBucket;
    private static MongoClient mongoClient;
    public static void main(String[] args) {
        Cluster cluster = CouchbaseCluster.create();
        defaultBucket = cluster.openBucket();
        migrate();
    }
    public static void migrate() {
        mongoClient = new MongoClient(Arrays.asList(new ServerAddress(
            "localhost", 27017)));
        MongoDatabase db = mongoClient.getDatabase("local");
        MongoCollection<Document> coll = db.getCollection("catalog");
        ViewResult result = defaultBucket.query(ViewQuery.from("catalog",
            "catalog_view"));
        for (ViewRow viewRow : result) {
            Document catalog = new Document();
            JsonDocument json = viewRow.document();
            JsonObject jsonObj = json.content();
            Set<java.lang.String> fieldNames = jsonObj.getNames();
            Iterator<String> iter = fieldNames.iterator();
            while (iter.hasNext()) {
                String fieldName = iter.next();
                String fieldValue = jsonObj.getString(fieldName);
                catalog = catalog.append(fieldName, fieldValue);
            }
            coll.insertOne(catalog);
        }
    }
}

```

7. To run the `MigrateCouchbaseToMongoDB` application right-click on `MigrateCouchbaseToMongoDB.java` in the Package Explorer and select Run As ► Java Application as shown in Figure 7-20.

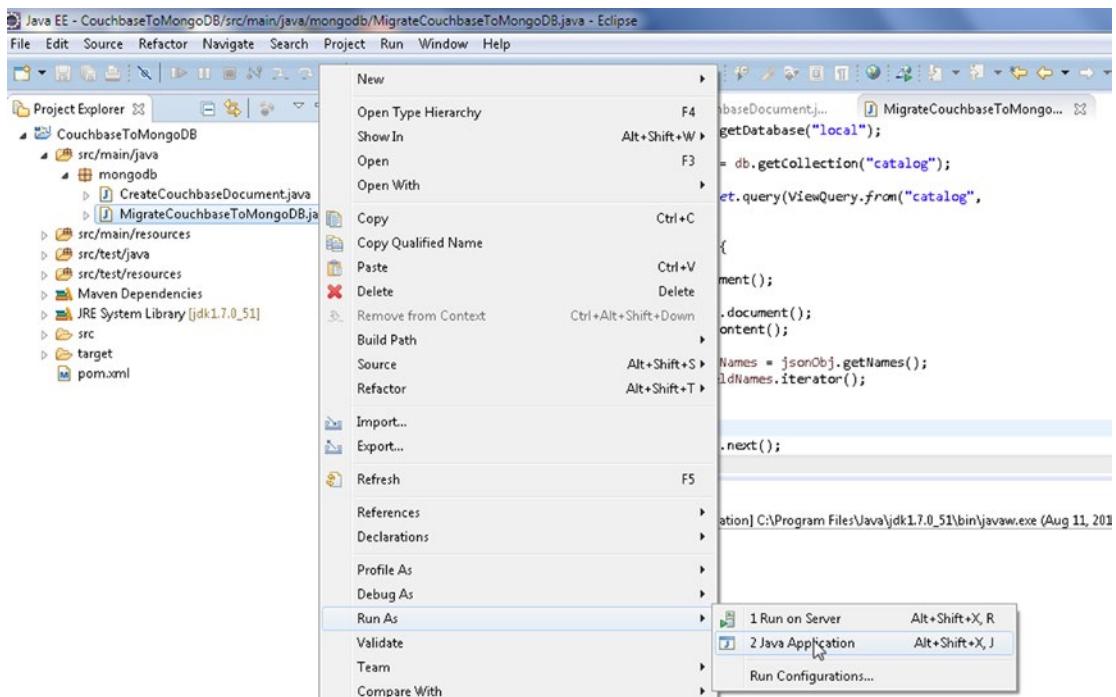


Figure 7-20. Running the *MigrateCouchbaseToMongoDB* Application

- The Couchbase Server documents get migrated to MongoDB. Subsequently run the following commands in Mongo shell.

```
>use local
>db.catalog.find()
```

The two JSON documents migrated to MongoDB get listed as shown in Figure 7-21.

```
> db.catalog.find()
{ "_id" : ObjectId("55ca40b9cd735001cc81cc55"), "journal" : "Oracle Magazine", "author" : "David A. Kelly", "edition" : "March April 2013", "title" : "Engineering as a Service", "publisher" : "Oracle Publishing" }
{ "_id" : ObjectId("55ca40b9cd735001cc81cc56"), "journal" : "Oracle Magazine", "author" : "Tom Haunert", "edition" : "March April 2013", "title" : "Quintessential and Collaborative", "publisher" : "Oracle Publishing" }
> -
```

Figure 7-21. Listing the Migrated Documents in Mongo Shell

Summary

Both Couchbase and MongoDB store documents as JSON with MongoDB having some advantages over Couchbase in document handling. In this chapter we migrated a Couchbase document to MongoDB. First, we created a JSON document in Couchbase in a Java application in Eclipse IDE. Subsequently we migrated the Couchbase document to MongoDB in another Java application. In the next chapter we shall migrate an Oracle Database table to MongoDB.

CHAPTER 8



Migrating Oracle Database

MongoDB is the leading NoSQL database. MongoDB stores documents using the BSON (binary JSON) data format, which is the most flexible of data models with provisions to create hierarchies of data structures. Oracle Database stores data in a fixed schema table format. The database models of the two databases are vastly different: MongoDB being based on Document store while Oracle Database is based on the fixed table format with two-dimensional matrices made of columns and rows. In this chapter we shall migrate an Oracle Database table to MongoDB Server. While a direct migration tool is not available for migrating from Oracle to MongoDB, the latter provides an import tool called the `mongoimport` tool for importing data from a CSV, TSV, or JSON file into a MongoDB database. We shall first export Oracle Database table to a CSV file. Subsequently we shall import the CSV file data into MongoDB Server using the `mongoimport` tool. This chapter covers the following topics.

- Overview of the `mongoimport` tool
- Setting up the environment
- Creating an Oracle Database table
- Exporting an Oracle Database table to a CSV file
- Importing data from a CSV file to MongoDB
- Displaying the JSON data in MongoDB

Overview of the `mongoimport` Tool

The `mongoimport` tool is used to transfer data from a file (CSV, TSV, or JSON) to MongoDB Server. The syntax for using the `mongoimport` tool is as follows.

```
mongoimport <options> <file>
```

Some of the options supported by `mongoimport` are discussed in Table 8-1.

Table 8-1. Options Supported by mongoimport Tool

Option	Description
-v , --verbose	Verbosity option for a detailed output. Include multiple times for more verbosity, for example, -vvv.
--quiet	Command output is not generated (data still transfers).
-h, --host	MongoDB host to connect to. Could include port in the format --host host:port.
--port	MongoDB port to connect to.
-u, --username	Username for authentication.
-p, --password	Password for authentication.
-d, --db	Database instance to use.
-c, --collection	Collection to use.
-f, --fields	Field names separated by comma.
--file	File to import from. If not specified, <code>stdin</code> is used. Could be a <code>.csv</code> file, <code>.json</code> file, or <code>.tsv</code> file.
--headerline	Use the first line in the input file as field list.
--jsonArray	Input source is JSON array.
--type	Input format to import. Value could be <code>json</code> , <code>csv</code> , or <code>tsv</code> . Default is <code>json</code> .
--drop	Drop collection before inserting documents.
--ignoreBlanks	Ignore fields with empty values in CSV and TSV.
--maintainInsertionOrder	Maintain insertion order.
--stopOnError	Stop importing at first insert/upsert error.
--upsert	Insert or update objects that already exist.
--writeConcern	Write concern options.
--numInsertionWorkers, -j	Number of insert operations to run concurrently.
--fieldFile	File with field names - one per line.

Setting Up the Environment

We need to download the following software for this chapter.

- Oracle Database 12c. Download from www.oracle.com/technetwork/database/enterprise-edition/downloads/index-092322.html.
- MongoDB Server (Version: 3.0.5).
- The `mongoimport` tool is installed with MongoDB Server and for Windows located in `C:\Program Files\MongoDB\Server\3.0\bin` directory.

As a reminder, when installing and configuring MongoDB Server create the `c:\data\db` directory.

Creating an Oracle Database Table

First, create an Oracle Database table WLSLOG using the following SQL script. The SQL script may be run from SQL*Plus.

```
CREATE TABLE OE.WLSLOG (ID VARCHAR2(255) PRIMARY KEY, TIME_STAMP VARCHAR2(255), CATEGORY
VARCHAR2(255), TYPE VARCHAR2(255), SERVERNAME VARCHAR2(255), CODE VARCHAR2(255), MSG
VARCHAR2(255));
INSERT INTO OE.WLSLOG (ID, TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog1','Apr-8-2014-7:06:16-PM-PDT','Notice','WebLogicServer','AdminServer','BEA-
000365','Server state changed to STANDBY');
INSERT INTO OE.WLSLOG (ID, TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog2','Apr-8-2014-7:06:17-PM-PDT','Notice','WebLogicServer','AdminServer','BEA-
000365','Server state changed to STARTING');
INSERT INTO OE.WLSLOG (ID,TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog3','Apr-8-2014-7:06:18-PM-PDT', 'Notice', 'WebLogicServer', 'AdminServer',
'BEA-000365', 'Server state changed to ADMIN');
INSERT INTO OE.WLSLOG (ID,TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog4','Apr-8-2014-7:06:19-PM-PDT', 'Notice', 'WebLogicServer', 'AdminServer',
'BEA-000365', 'Server state changed to RESUMING');
INSERT INTO OE.WLSLOG (ID,TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog5','Apr-8-2014-7:06:20-PM-PDT', 'Notice', 'WebLogicServer', 'AdminServer',
'BEA-000361', 'Started WebLogic AdminServer');
INSERT INTO OE.WLSLOG (ID,TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog6','Apr-8-2014-7:06:21-PM-PDT', 'Notice', 'WebLogicServer', 'AdminServer',
'BEA-000365', 'Server state changed to RUNNING');
INSERT INTO OE.WLSLOG (ID,TIME_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, MSG) values
('catalog7','Apr-8-2014-7:06:22-PM-PDT', 'Notice', 'WebLogicServer', 'AdminServer',
'BEA-000360', 'Server started in RUNNING mode');
```

Exporting an Oracle Database Table to a CSV File

Next, export the Oracle Database table to a CSV file. Run the following SQL script in SQL*Plus to select data from the OE.WLSLOG table and export to a wlslog.csv file.

```
set pagesize 0 linesize 500 trimspool on feedback off echo off
select ID || ',' || TIME_STAMP || ',' || CATEGORY || ',' || TYPE || ',' || SERVERNAME || ',' ||
CODE || ',' || MSG from OE.WLSLOG;
spool wlslog.csv
/
spool off
```

When the SQL script is run as shown in Figure 8-1, data is exported to the wlslog.csv file.

```

SQL> set pagesize 0 linesize 500 trimspool on feedback off echo off
SQL> select ID || ',' || TIME_STAMP || ',' || CATEGORY || ',' || TYPE || ',' ||
SERVERNAME || ',' || CODE || ',' || MSG from OE.WLSLOG;
catalog1,Apr-8-2014-7:06:16-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to STANDBY
catalog2,Apr-8-2014-7:06:17-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to STARTING
catalog3,Apr-8-2014-7:06:18-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to ADMIN
catalog4,Apr-8-2014-7:06:19-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to RESUMING
catalog5,Apr-8-2014-7:06:20-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000361,
Started WebLogic AdminServer
catalog6,Apr-8-2014-7:06:21-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to RUNNING
catalog7,Apr-8-2014-7:06:22-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000360,
Server started in RUNNING mode
SQL> spool wlslog.csv
SQL> /
catalog1,Apr-8-2014-7:06:16-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to STANDBY
catalog2,Apr-8-2014-7:06:17-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to STARTING
catalog3,Apr-8-2014-7:06:18-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to ADMIN
catalog4,Apr-8-2014-7:06:19-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to RESUMING
catalog5,Apr-8-2014-7:06:20-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000361,
Started WebLogic AdminServer
catalog6,Apr-8-2014-7:06:21-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,
Server state changed to RUNNING
catalog7,Apr-8-2014-7:06:22-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000360,
Server started in RUNNING mode
SQL> spool off
SQL>

```

Figure 8-1. Exporting Oracle Database Table to CSV File

Remove the leading SQL> / and trailing SQL> spool off from the output exported to save the following as the wlslog.csv file. The leading and trailing lines are to be removed because the input to the mongoimport tool should be a CSV file.

```

catalog1,Apr-8-2014-7:06:16-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,Server state
changed to STANDBY
catalog2,Apr-8-2014-7:06:17-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,Server state
changed to STARTING
catalog3,Apr-8-2014-7:06:18-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,Server state
changed to ADMIN
catalog4,Apr-8-2014-7:06:19-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,Server state
changed to RESUMING
catalog5,Apr-8-2014-7:06:20-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000361,Started
WebLogic AdminServer
catalog6,Apr-8-2014-7:06:21-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000365,Server state
changed to RUNNING
catalog7,Apr-8-2014-7:06:22-PM-PDT,Notice,WebLogicServer,AdminServer,BEA-000360,Server
started in RUNNING mode

```

Importing Data from a CSV File to MongoDB

In this section we shall transfer data from the `wlslog.csv` file to MongoDB Server using the `mongoimport` tool.

1. Start the MongoDB server with the following command.

```
>mongod
```

MongoDB gets started on localhost on port 27017 as shown in Figure 8-2.

```
C:\MongoDB>mongod
2015-08-04T15:41:42.592-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-08-04T15:41:42.780-0700 I JOURNAL [initandlisten] journal dir=C:\data\db\j
ournal
2015-08-04T15:41:42.781-0700 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-08-04T15:41:42.873-0700 I JOURNAL [durability] Durability thread started
2015-08-04T15:41:42.875-0700 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-08-04T15:41:43.004-0700 I CONTROL [initandlisten] MongoDB starting : pid=9
032 port=27017 dbpath=C:\data\db\ 64-bit host=dvohra-PC
2015-08-04T15:41:43.004-0700 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-08-04T15:41:43.004-0700 I CONTROL [initandlisten] db version v3.0.5
2015-08-04T15:41:43.005-0700 I CONTROL [initandlisten] git version: 8bc4ae20708
dbb493cb09338d9e7be6698e4a3a3
2015-08-04T15:41:43.005-0700 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_49
2015-08-04T15:41:43.005-0700 I CONTROL [initandlisten] allocator: tcmalloc
2015-08-04T15:41:43.005-0700 I CONTROL [initandlisten] options: <>
2015-08-04T15:41:44.356-0700 I NETWORK [initandlisten] waiting for connections
on port 27017
-
```

Figure 8-2. Starting MongoDB Server

2. Run the `mongoimport` tool to transfer data from the `wlslog.csv` file to the MongoDB Server. The command options are listed in Table 8-2.

Table 8-2. Command Parameters Used for the `mongoimport` Tool Command

Option	Value	Description
--db	wls	MongoDB database instance. The database is not required to be created prior to running the <code>mongoimport</code> tool and gets created when the import toll is run.
--collection	wlslog	Collection name. The collection is also not required to be created prior to running the import tool.
--type	csv	Type of input data is CSV.
--fields	ID,TIME_STAMP,CATEGORY,TYPE, SERVERNAME,CODE,MSG	Input fields.
--file	wlslog.csv	Input file.

3. Run the following mongoimport command.

```
mongoimport --db wls --collection wlslog --type csv --fields ID,TIME_STAMP,
CATEGORY,TYPE,SERVERNAME,CODE,MSG --file wlslog.csv
```

As the output indicates, data gets transferred to MongoDB Server from the `wlslog.csv` file as shown in Figure 8-3. For the 7 lines of input data 7 documents get created in MongoDB Server.

```
C:\MongoDB>mongoimport --db wls --collection wlslog --type csv --fields ID,TIME_STAMP,CATEGORY,TYPE,SERVERNAME,CODE,MSG --file wlslog.csv
2015-08-04T15:57:33.616-0700    connected to: localhost
2015-08-04T15:57:33.620-0700    imported 7 documents
C:\MongoDB>
```

Figure 8-3. Running the `mongoimport` Tool Command

Displaying the JSON Data in MongoDB

The following steps show you how to display the JSON data:

1. Start the Mongo shell with the following command.

```
>mongo
```

2. To use the `wls` database run the following Mongo shell command.

```
>use wls
```

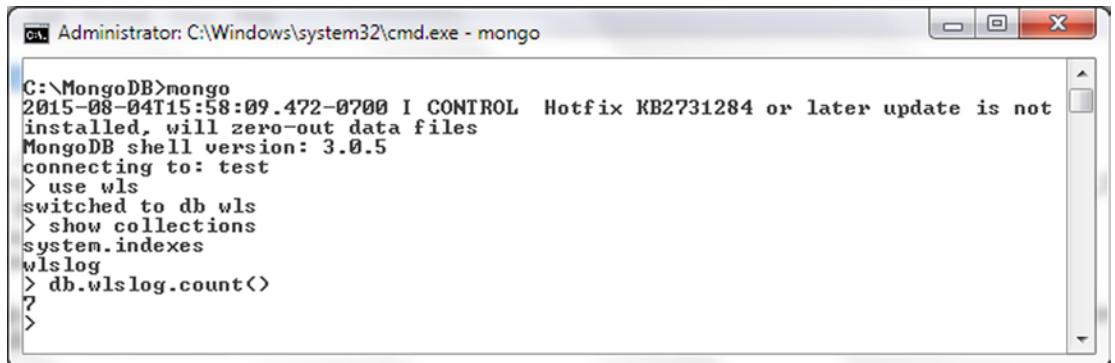
3. To list the collections run the following Mongo shell command.

```
>show collections
```

4. To output the document count run the following Mongo shell command.

```
>db.wlslog.count()
```

The output from the preceding commands is shown in Figure 8-4.



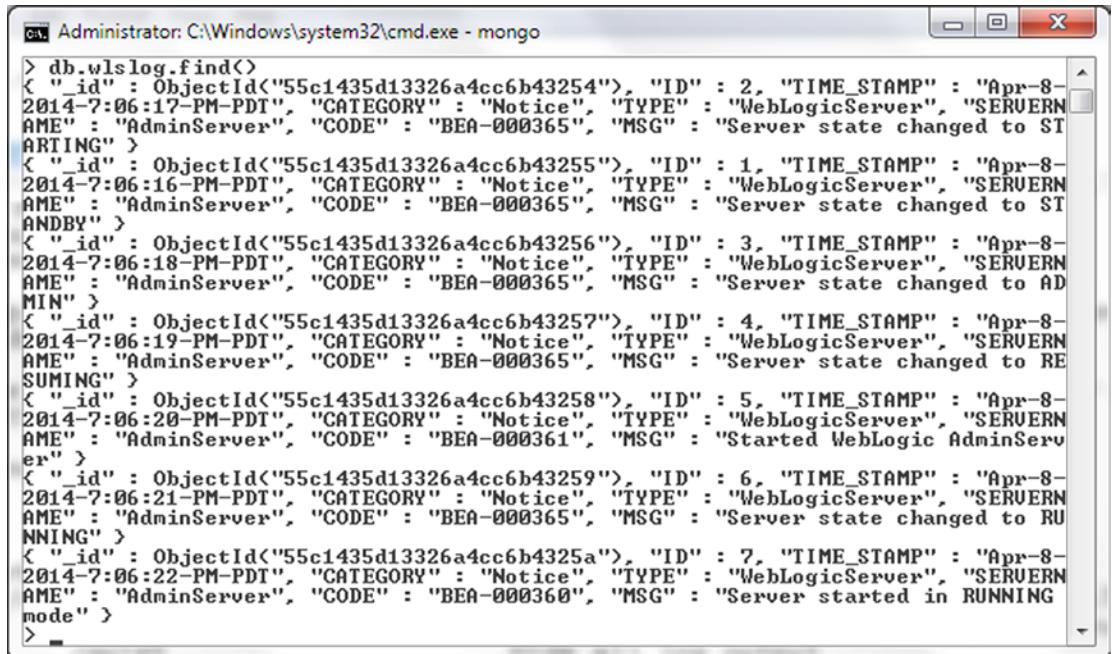
```
C:\MongoDB>mongo
2015-08-04T15:58:09.472-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
> use wls
switched to db wls
> show collections
system.indexes
wlslog
> db.wlslog.count()
7
>
```

Figure 8-4. The Document Count is 7

- To list the documents imported to MongoDB Server run the following Mongo shell command.

```
>db.wlslog.find()
```

The documents imported to MongoDB Server get listed as shown in Figure 8-5.



```
> db.wlslog.find()
[{"_id": ObjectId("55c1435d13326a4cc6b43254"), "ID": 2, "TIME_STAMP": "Apr-8-2014-7:06:17-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000365", "MSG": "Server state changed to STARTING"}, {"_id": ObjectId("55c1435d13326a4cc6b43255"), "ID": 1, "TIME_STAMP": "Apr-8-2014-7:06:16-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000365", "MSG": "Server state changed to STANDBY"}, {"_id": ObjectId("55c1435d13326a4cc6b43256"), "ID": 3, "TIME_STAMP": "Apr-8-2014-7:06:18-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000365", "MSG": "Server state changed to ADMIN"}, {"_id": ObjectId("55c1435d13326a4cc6b43257"), "ID": 4, "TIME_STAMP": "Apr-8-2014-7:06:19-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000365", "MSG": "Server state changed to RESUMING"}, {"_id": ObjectId("55c1435d13326a4cc6b43258"), "ID": 5, "TIME_STAMP": "Apr-8-2014-7:06:20-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000361", "MSG": "Started WebLogic AdminServer"}, {"_id": ObjectId("55c1435d13326a4cc6b43259"), "ID": 6, "TIME_STAMP": "Apr-8-2014-7:06:21-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000365", "MSG": "Server state changed to RUNNING"}, {"_id": ObjectId("55c1435d13326a4cc6b4325a"), "ID": 7, "TIME_STAMP": "Apr-8-2014-7:06:22-PM-PDT", "CATEGORY": "Notice", "TYPE": "WebLogicServer", "SERVERNAME": "AdminServer", "CODE": "BEA-000360", "MSG": "Server started in RUNNING mode"}]
```

Figure 8-5. The 7 Documents Transferred to MongoDB Server

Summary

In this chapter we transferred Oracle Database table data to MongoDB Server. First, we created an Oracle Database table. As a direct data transfer tool is not available, first we exported the Oracle Database table to a CSV file. Subsequently the CSV file data is transferred to MongoDB Server using the MongoDB `mongoimport` tool. In the next chapter we shall use Kundera, a JPA 2.0 compliant Object-Datastore Mapping library for NoSQL datastores.

CHAPTER 9



Using Kundera with MongoDB

The Java Persistence API (JPA) is the Java API for persistence management and object/relational mapping in Java EE/Java SE environment with which a Java domain model is used to manage a relational database. JPA also provides a query language API with the Query interface for static and dynamic queries. JPA is designed primarily for relational databases, and Kundera is a JPA 2.0-compliant Object-Datastore Mapping library for NoSQL datastores. Kundera also supports relational databases and provides NoSQL datastore-specific configurations for MongoDB and some other NoSQL databases: Apache Cassandra, and HBase. Using the kundera-mongo library in the domain model MongoDB may be accessed using the JPA. In this chapter we shall access MongoDB with the kundera-mongo module and run CRUD (Create, Read, Update, and Delete) operations on MongoDB, covering the following topics:

- Setting up the environment
- Creating a MongoDB collection
- Creating a Maven project in Eclipse
- Creating a JPA entity class
- Configuring JPA in the `persistence.xml` configuration file
- Creating a JPA client class
- Running JPA CRUD operations
- The Kundera-Mongo JPA Client class
- Installing the Maven project
- Running the Kundera-Mongo JPA Client class
- Invoking the `KunderaClient` methods

Setting Up the Environment

We shall need the following software for this chapter.

- Eclipse IDE for Java EE Developers from www.eclipse.org/downloads/.
- MongoDB 3.0.2 (or a later version) Windows binaries `mongodb-win32-x86_64-3.0.2-signed.msi` from www.mongodb.org/dl/win32/x86_64. Double-click on the `mongodb-win32-x86_64-3.0.2-signed` file to install MongoDB 3.0.2. Add the `bin` directory, for example `C:\Program Files\MongoDB\Server\3.0\bin`, to the PATH environment variable.
- Java 7 from www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

The MongoDB versions supported by Kundera are 2.6.3+ & 3.0.2. Create a directory `C:\data\db` for the MongoDB data if not already created for an earlier chapter. Start MongoDB with the following command from a command shell.

```
>mongod
```

MongoDB gets started waiting for connections on `localhost:27017`.

Creating a MongoDB Collection

We need to create a MongoDB collection in which to store the documents. Start the Mongo shell with the `mongo` command

```
>mongo
```

Create a collection called `catalog` with the following commands in Mongo shell. The first command sets the database to `local`. The second command drops the `catalog` collection.

```
>use local
>db.catalog.drop()
>db.createCollection("catalog")
```

The `catalog` collection gets created as shown in Figure 9-1.

```
> use local
switched to db local
> db.catalog.drop()
true
> db.createCollection("catalog")
{ "ok" : 1 }
>
```

Figure 9-1. Creating MongoDB Collection

Creating a Maven Project in Eclipse

The kundera-mongo library is available as a Maven dependency. We shall use a Maven project to access MongoDB with the kundera-mongo library for which to create a Maven project in Eclipse IDE.

1. Select File > New > Other in Eclipse IDE.
2. Select Maven > Maven Project in the New window as shown in Figure 9-2.
Click on Next.

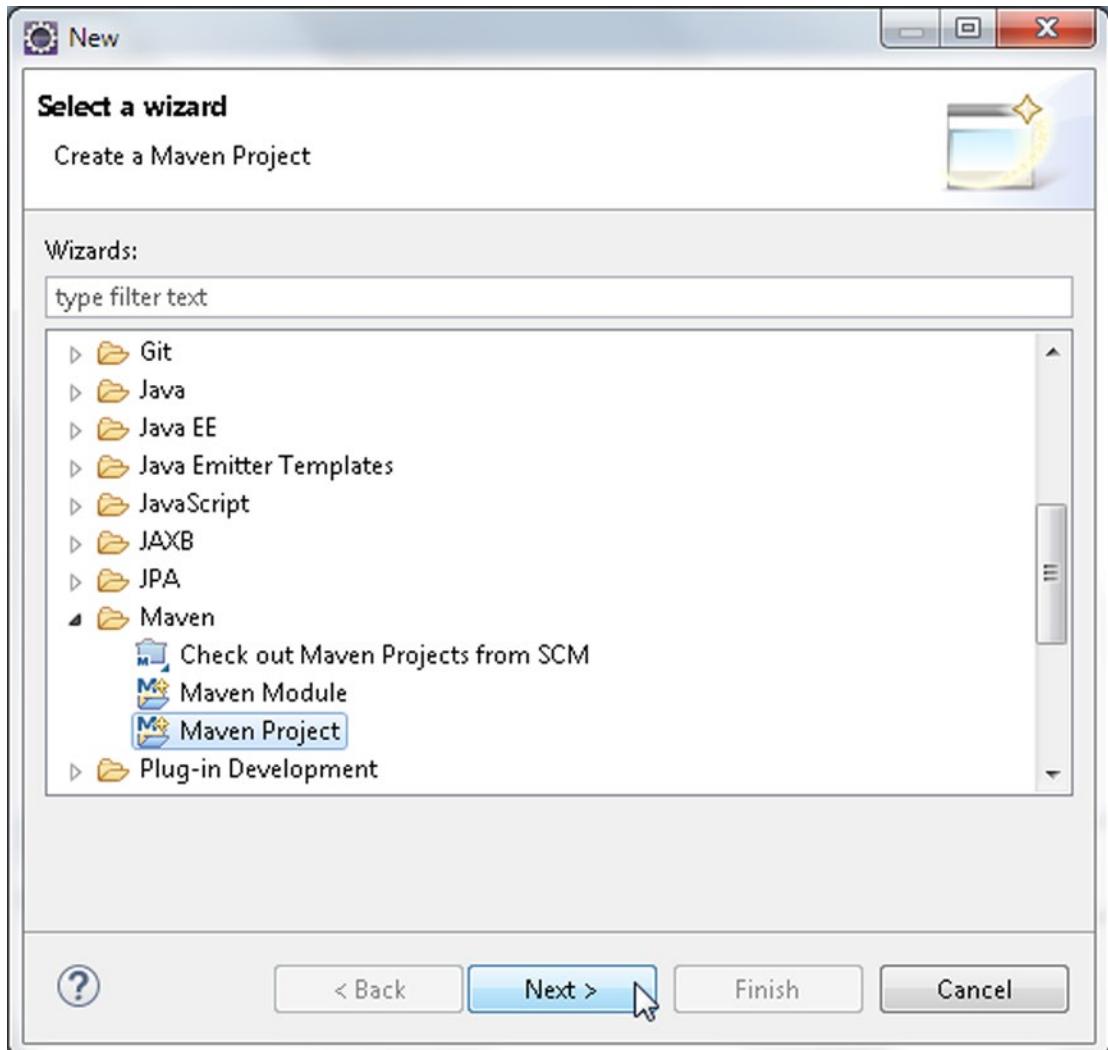


Figure 9-2. Selecting Maven > Maven Project

3. In New Maven Project wizard select the Create a simple project check box. Also select the Use default Workspace location check box as shown in Figure 9-3. Click on Next.

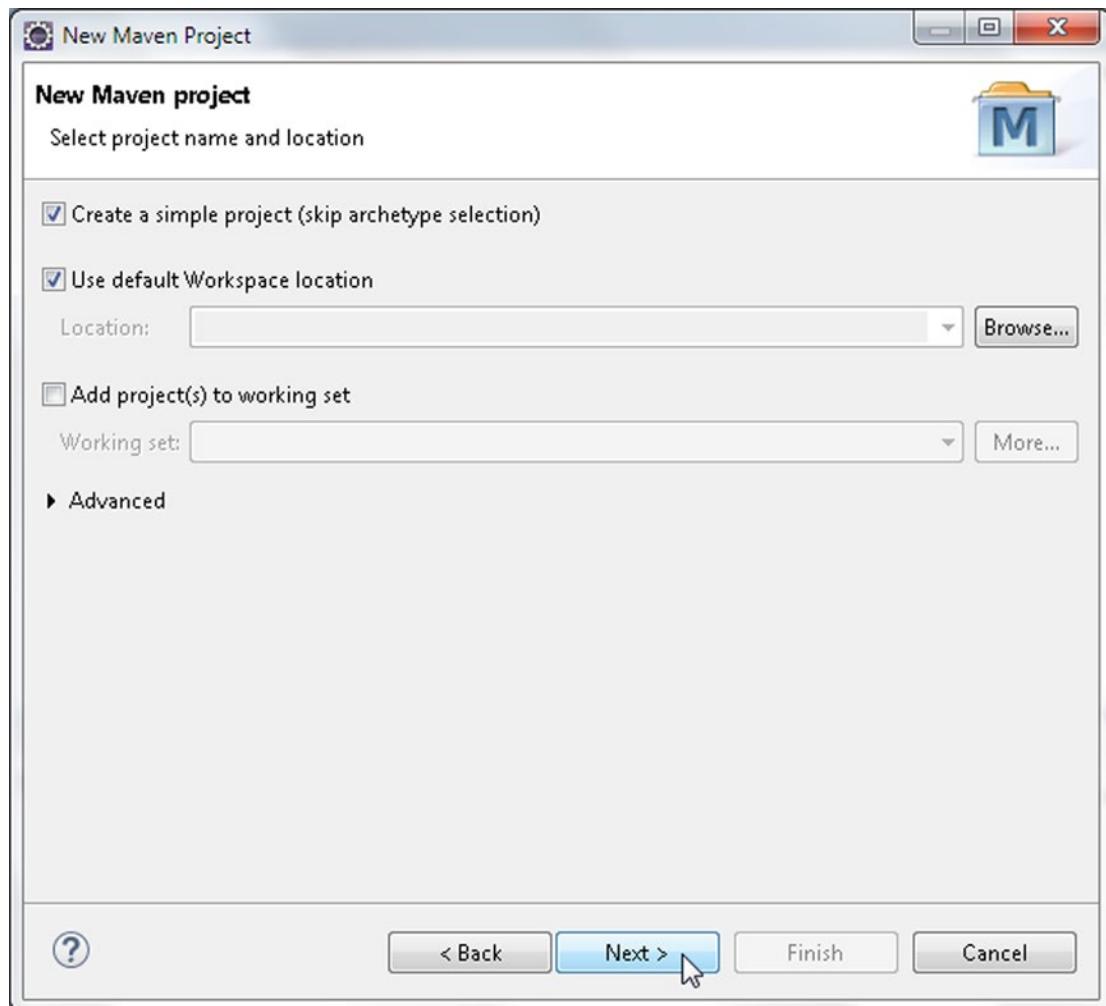


Figure 9-3. New Maven Project Wizard

4. In the New Maven wizard's Configure project window, select the following values and click on Finish as shown in Figure 9-4.

- Group Id: com.kundera.mongodb
- Artifact Id: KunderaMongoDB
- Version: 1.0.0
- Packaging: jar
- Name: KunderaMongoDB

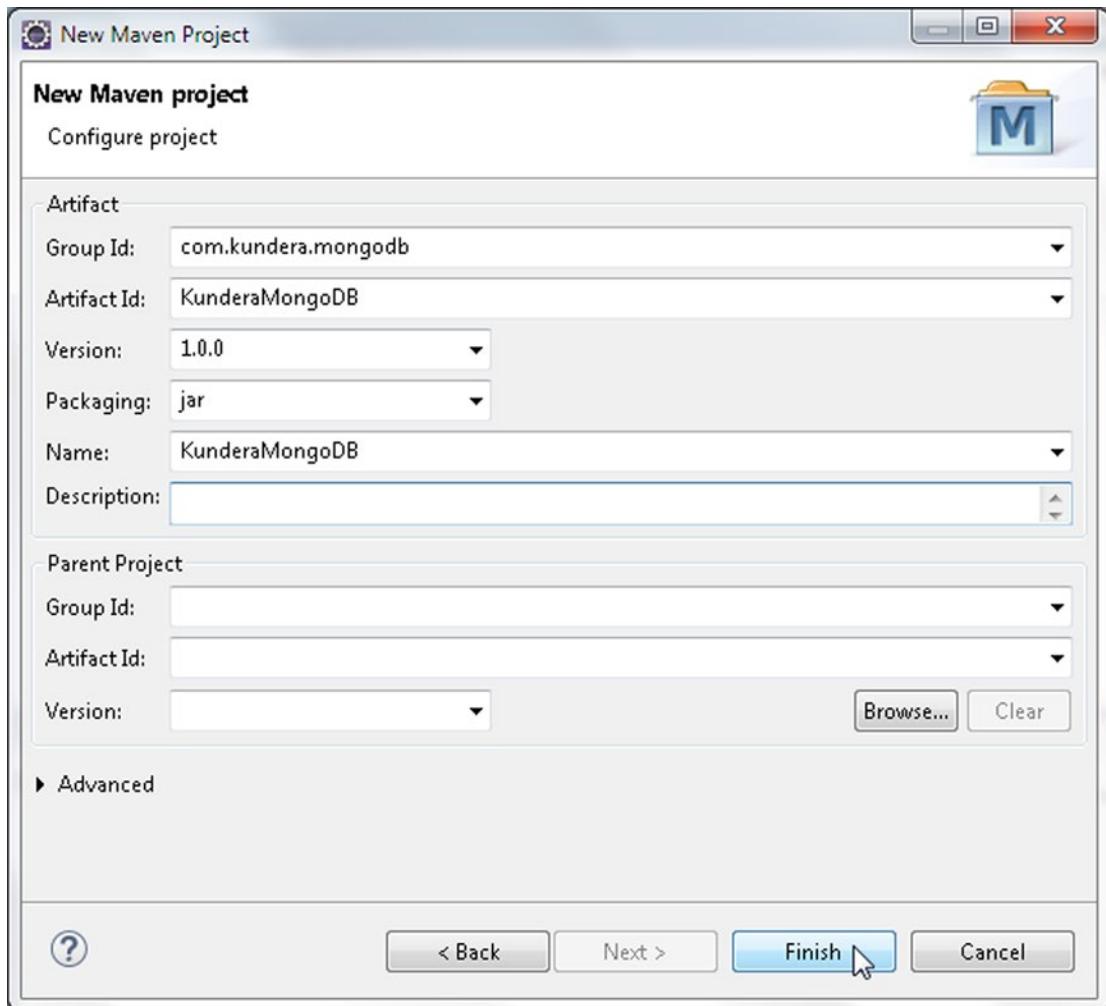


Figure 9-4. Configuring Maven Project

A Maven project for Kundera Mongo gets created as shown in the Package Explorer in Figure 9-5.

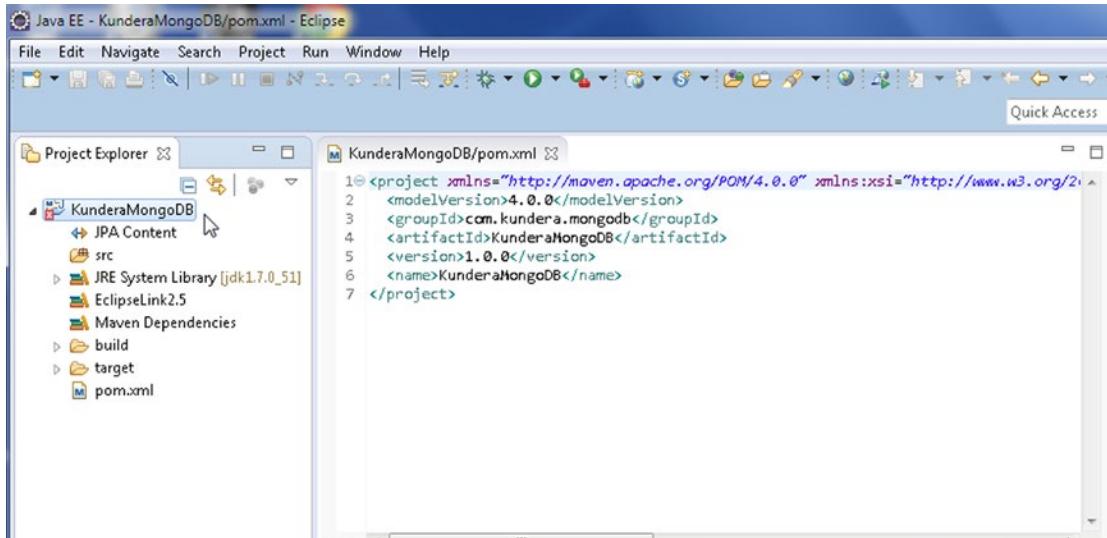


Figure 9-5. Maven Project KunderaMongoDB

5. Next, modify the `pom.xml` to add dependencies for the kundera-mongo library and EclipseLink. As we shall be using JPA, EclipseLink is required.

```

<dependencies>
  <dependency>
    <groupId>com.impetus.client</groupId>
    <artifactId>kundera-mongo</artifactId>
    <version>2.9</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.6.0</version>
  </dependency>
</dependencies>

```

6. In the build configuration specify the `maven-compiler-plugin` plug-in to compile the Maven project and the `exec-maven-plugin` plug-in to run the Maven client class. For the `exec-maven-plugin` plug-in specify the main class to run in the `<configuration/>` as the Kundera Client class `kundera.KunderaClient`, which we shall create later in the chapter.

```

<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.2.1</version>
            <configuration>
                <mainClass>kundera.KunderaClient</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>

```

The `pom.xml` is listed:

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.kundera.mongodb</groupId>
  <artifactId>KunderaMongoDB</artifactId>
  <version>1.0.0</version>
  <name>KunderaMongoDB</name>
  <dependencies>
    <dependency>
      <groupId>com.impetus.client</groupId>
      <artifactId>kundera-mongo</artifactId>
      <version>2.9</version>
    </dependency>
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>eclipselink</artifactId>
      <version>2.6.0</version>
    </dependency>
  </dependencies>
</project>

```

```

</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.2.1</version>
            <configuration>
                <mainClass>kundera.KunderaClient</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>

```

Creating a JPA Entity Class

The domain model for a JPA object/relational mapping application is defined in a JPA entity class. The domain model class is just a *plain old Java object* (POJO) that describes the Java object entity to be persisted, the object properties, and the MongoDB database and collection to persist to. In this section we shall create a JPA entity class for object/relational mapping using Kundera and MongoDB databases. MongoDB, though not a relational database, can be used with object/relational mapping using the Kundera library, which supports some NoSQL databases.

1. Select File ▶ New ▶ Other.
2. In the New window, select Java ▶ Class and click on Next as shown in Figure 9-6.

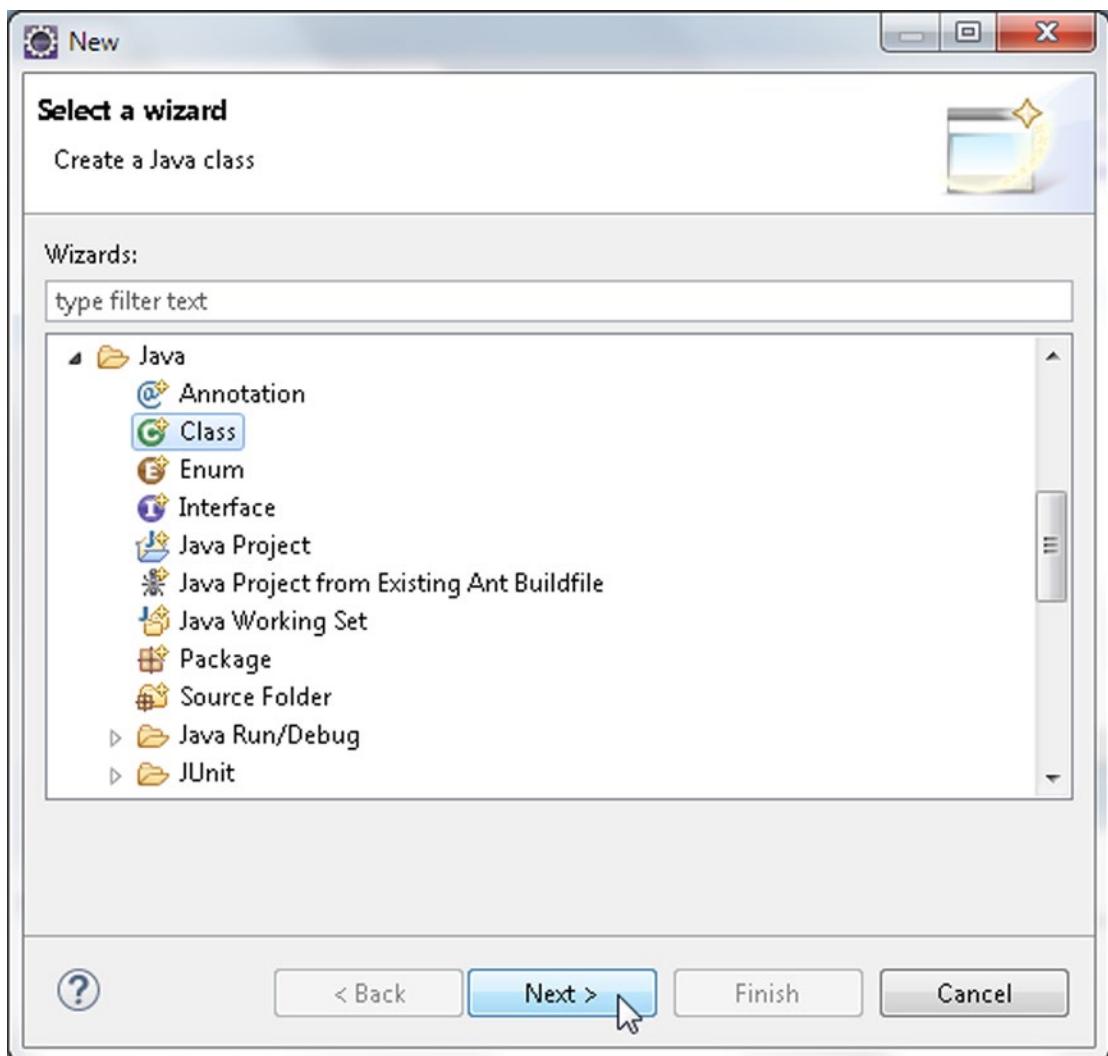


Figure 9-6. Selecting Java ► Class

3. In New Java Class wizard assign the following values as shown in Figure 9-7. Click on Finish.
 - Source folder: KunderaMongoDB/src/main/java
 - Package: kundera
 - Name: Catalog

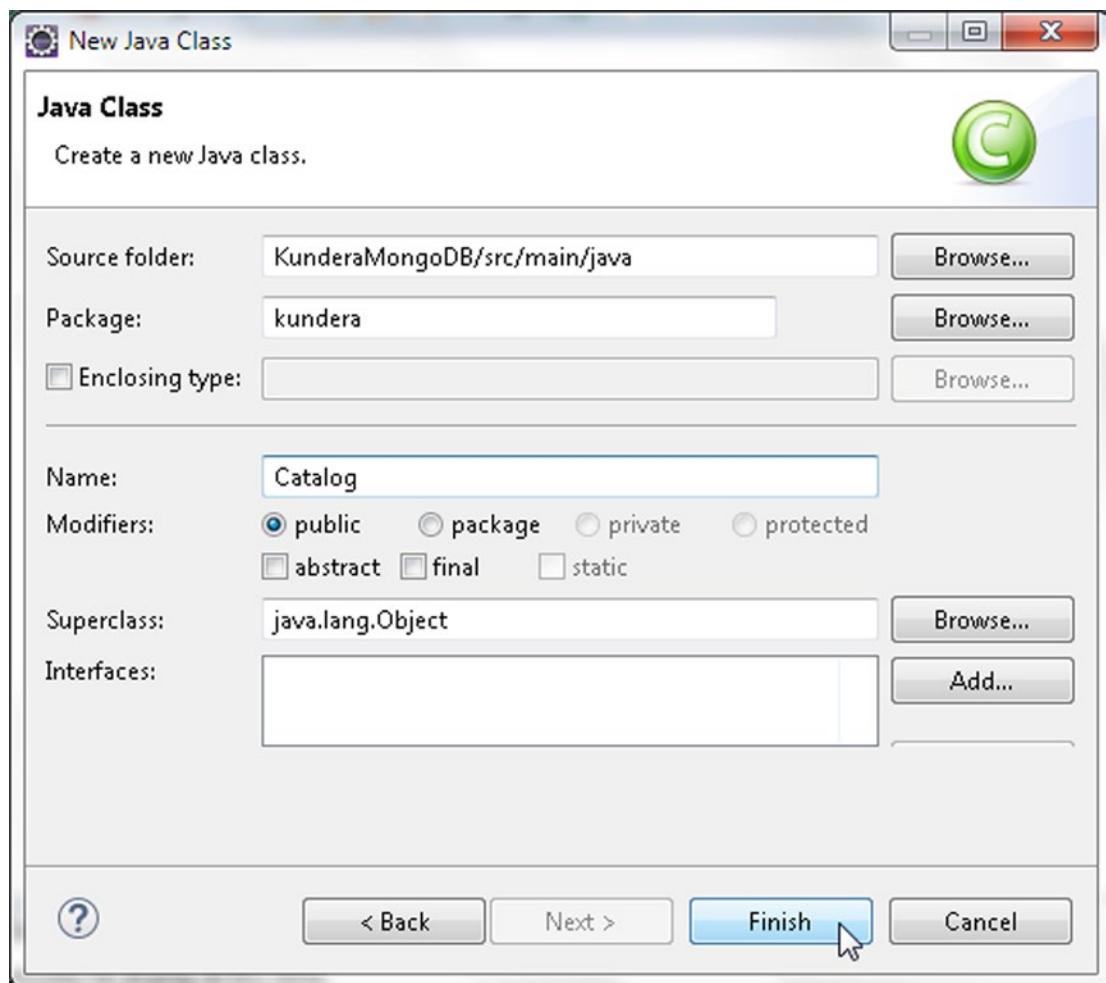


Figure 9-7. Creating Entity Class Catalog

The Java class `kundera.Catalog` gets added to the Maven project as shown in Figure 9-8.

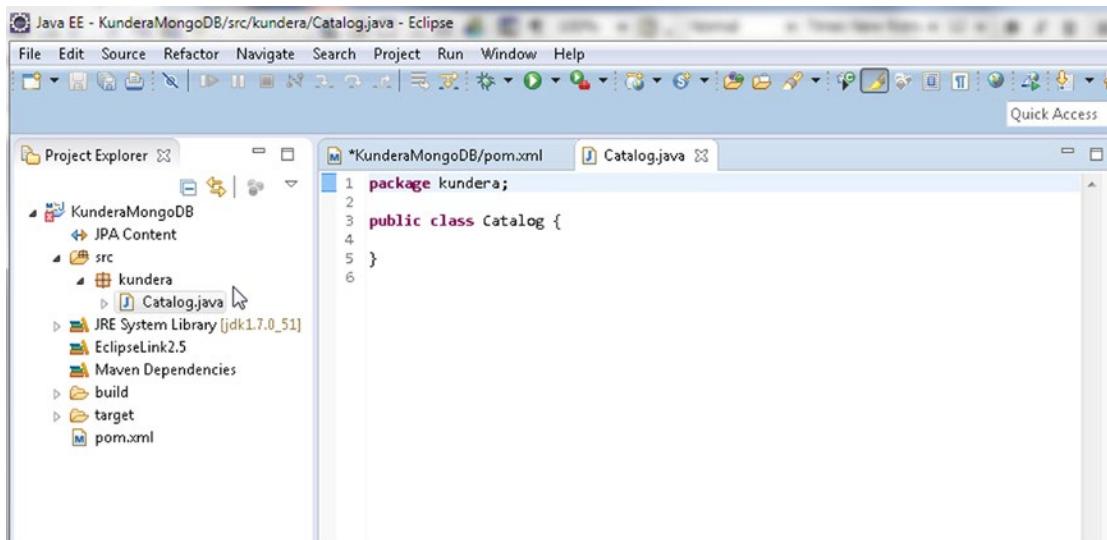


Figure 9-8. Entity Class Catalog

4. Annotate the Catalog class with @Entity annotation to indicate that the class is a JPA entity class. By default the entity name is the same as the entity class name. Annotate the class with @Table to indicate the MongoDB collection name and schema. The table name is the collection name catalog. The schema is in database@persistence-unit format. For the local database and the kundera persistence unit name, which we shall configure in the next section, the schema is local@kundera.

```
@Entity
@Table(name = "catalog", schema = "local@kundera")
```

5. The entity class implements the Serializable interface to serialize a cache enabled entity bean to a cache when persisted to a database. To associate a version number with a serializable class by serialization runtime specify a serialVersionUID variable.

```
private static final long serialVersionUID = 1L;
```

6. Annotate the catalogId field with the @Id annotation to indicate that the field is the primary key of the entity. Specify the document field _id as the primary key column using the @Column annotation.

```
@Id
@Column(name = "_id")
private String catalogId;
```

The field annotated with @Id must be one of the following types: Java primitive type (such as int, double), any primitive wrapper type (such as Integer, Double), String, java.util.Date, java.sql.Date, java.math.BigDecimal, or java.math.BigInteger.

7. Add fields `journal`, `publisher`, `edition`, `title`, and `author` and annotate them with the `@Column` annotation to specify the mapping of the fields to columns in the document.

```
@Column(name = "journal")
private String journal;
@Column(name = "publisher")
private String publisher;
@Column(name = "edition")
private String edition;
@Column(name = "title")
private String title;
@Column(name = "author")
private String author;
```

8. Add get/set accessor methods for each of the fields. The JPA Entity class is listed below.

```
package kundera;

import java.io.Serializable;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Catalog
 */
@Entity
@Table(name = "catalog", schema = "local@kundera")
public class Catalog implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "_id")
    private String catalogId;

    public Catalog() {
        super();
    }

    @Column(name = "journal")
    private String journal;

    @Column(name = "publisher")
    private String publisher;

    @Column(name = "edition")
    private String edition;
```

```
@Column(name = "title")
private String title;

@Column(name = "author")
private String author;

public String getCatalogId() {
    return catalogId;
}

public void setCatalogId(String catalogId) {
    this.catalogId = catalogId;
}

public String getJournal() {
    return journal;
}

public void setJournal(String journal) {
    this.journal = journal;
}

public String getPublisher() {
    return publisher;
}

public void setPublisher(String publisher) {
    this.publisher = publisher;
}

public String getEdition() {
    return edition;
}

public void setEdition(String edition) {
    this.edition = edition;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}
```

Configuring JPA in the persistence.xml Configuration File

In this section we shall create a META-INF/persistence.xml configuration file in the src/main/resources folder in the Maven project. We shall configure the object/relational mapping in the persistence.xml configuration file. Kundera supports some properties that are specified in persistence.xml using the <property/> tag, common to all NoSQL datastores it supports. These common properties are discussed in Table 9-1.

Table 9-1. Kundera Persistence Configuration Properties

Property	Description	Required/Optional
kundera.nodes	Nodes/s on which NoSQL server is running.	Required
kundera.port	NoSQL database port.	Required
kundera.keyspace	NoSQL database keyspace.	Required
kundera.dialect	The NoSQL database dialect to determine the persistence provider. Valid values are: cassandra, mongodb, and hbase.	Required
kundera.client.lookup.class	NoSQL database-specific client class for low-level datastore operations.	Required
kundera.cache.provider.class	The L2 cache implementation class.	Required
kundera.cache.config.resource	File containing L2 cache implementation.	Required
kundera.ddl.auto.prepare	Specifies an option to automatically generate schema and tables for all entities. Valid options are: create: Drops, if one exists, the schema and creates schema/tables based on entity definitions. create-drop: Same as create and in addition drops schema after operation ends. update: Updates schema/tables based on entity definitions. validate: Validates schema/table based on entity definitions and throws SchemaGenerationException if validation fails.	Optional
kundera.pool.size.max.active	Upper limit on the number of object instances managed by the pool per node.	Optional
kundera.pool.size.max.idle	Upper limit on the number of idle object instances in the pool.	Optional
kundera.pool.size.min.idle	Minimum number of idle object instances in the pool.	Optional
kundera.pool.size.max.total	Upper limit on the total number of object instances in the pool from all nodes combined.	Optional
index.home.dir	If Lucene indexes are chosen instead of the inbuilt secondary indexes, directory path to store Lucene indexes.	Optional

(continued)

Table 9-1. (continued)

Property	Description	Required/Optional
kundera.client.property	Name of the NoSQL database-specific configuration file, which must be in the classpath.	Optional
kundera.batch.size	Batch size in integer for bulk insert/update.	Optional
kundera.username	Username to authenticate Cassandra and MongoDB.	Optional
kundera.password	Password to authenticate Cassandra and MongoDB.	Optional

9. In the `persistence.xml` for the Maven project specify persistence-unit name as `kundera`.
10. Add a `<provider/>` element set to `com.impetus.kundera.KunderaPersistence`.
11. Specify the JPA entity class to `kundera.Catalog` in the `<class/>` element.
12. Add `<property/>` tags grouped as subelements of the `<properties/>` tag. Add the properties discussed in Table 9-2.

Table 9-2. Configured Kundera Persistence Properties

Property	Description	Value
<code>kundera.nodes</code>	MongoDB host name.	localhost or 127.0.0.1
<code>kundera.port</code>	MongoDB port.	27017
<code>kundera.keyspace</code>	MongoDB database.	local
<code>kundera.dialect</code>	Kundera Dialect. Used by Kundera to determine persistence provider implementation.	mongodb
<code>kundera.ddl.auto.prepare</code>	Used by Kundera to automatically generate schema and entities in a persistence unit.	create
<code>kundera.client.lookup.class</code>	Used by Kundera to find low-level dialect classes to perform operations on the MongoDB.	<code>com.impetus.client.mongodb.MongoDBClientFactory</code>
<code>kundera.annotations.scan.package</code>	The package to scan for JPA entities.	<code>kundera</code>

The `persistence.xml` configuration file is listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/
  persistence/persistence_2_1.xsd">
  <persistence-unit name="kundera">
    <provider>com.impetus.kundera.KunderaPersistence</provider>
    <class>kundera.Catalog</class>
    <properties>
      <property name="kundera.nodes" value="127.0.0.1" />
      <property name="kundera.port" value="27017" />
      <property name="kundera.keyspace" value="local" />
      <property name="kundera.dialect" value="mongodb" />
      <property name="kundera.ddl.auto.prepare" value="create" />
      <property name="kundera.client.lookup.class"
        value="com.impetus.client.mongodb.MongoDBClientFactory" />
      <property name="kundera.annotations.scan.package" value="kundera" />
    </properties>
  </persistence-unit>
</persistence>
```

Some NoSQL database-specific properties may also be specified in a separate XML configuration file. The following (Table 9-3) MongoDB-specific properties are supported.

Table 9-3. MongoDB-Specific Configuration Properties

Property	Description
<code>read.preference</code>	The read preference. Whether the primary or secondary server should be read from.
<code>socket.timeout</code>	The socket timeout is the time period in milliseconds that Kundera waits for connection from MongoDB before timing out.

13. For example, to configure MongoDB-specific properties add the following property for the MongoDB-specific configuration file in `persistence.xml`.

```
<property name="kundera.client.property" value="kundera-mongo.xml" />
```

The name of the MongoDB-specific configuration file, `kundera-mongo.xml`. A sample MongoDB-specific configuration file is listed.

```
<?xml version="1.0" encoding="UTF-8"?>
<clientProperties>
  <datastores>
    <dataStore>
      <name>mongo</name>
      <connection>
        <properties>
```

```

<property name="read.preference" value="secondary"></property>
<property name="socket.timeout" value="50000"></property>
</properties>
<servers>
    <server>
        <host>192.160.140.160</host>
        <port>27017</port>
    </server>
    <server>
        <host>192.161.141.161</host>
        <port>27018</port>
    </server>
</servers>
</connection>
</dataStore>
</datastores>
</clientProperties>

```

We have not used any MongoDB-specific configuration file.

14. Next, create the `persistence.xml` configuration file in the `src/main/resources/META-INF` folder. The `META-INF` folder is not created when a new Maven project is created. To add the `META-INF` folder right-click on the `resources` folder and select `New > Folder` as shown in Figure 9-9.

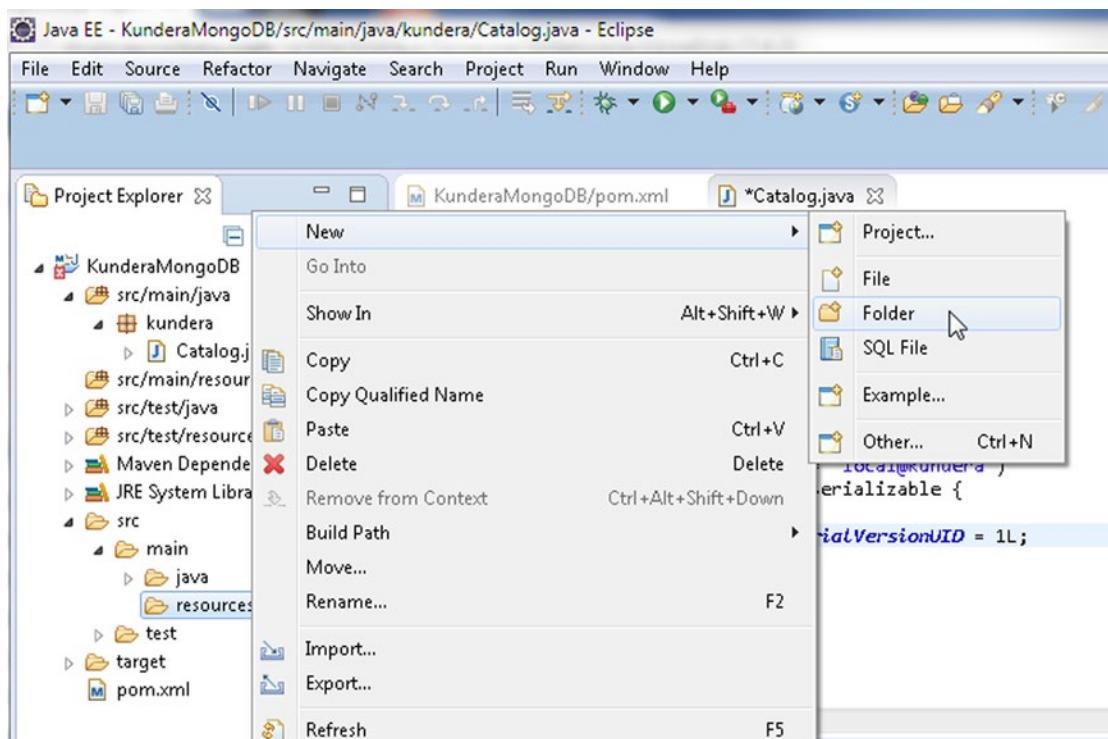


Figure 9-9. Creating a new `Folder` in `resources` Folder

15. In New Folder wizard select the `src/main/resources` folder and specify Folder name as `META-INF` and click on Finish as shown in Figure 9-10.

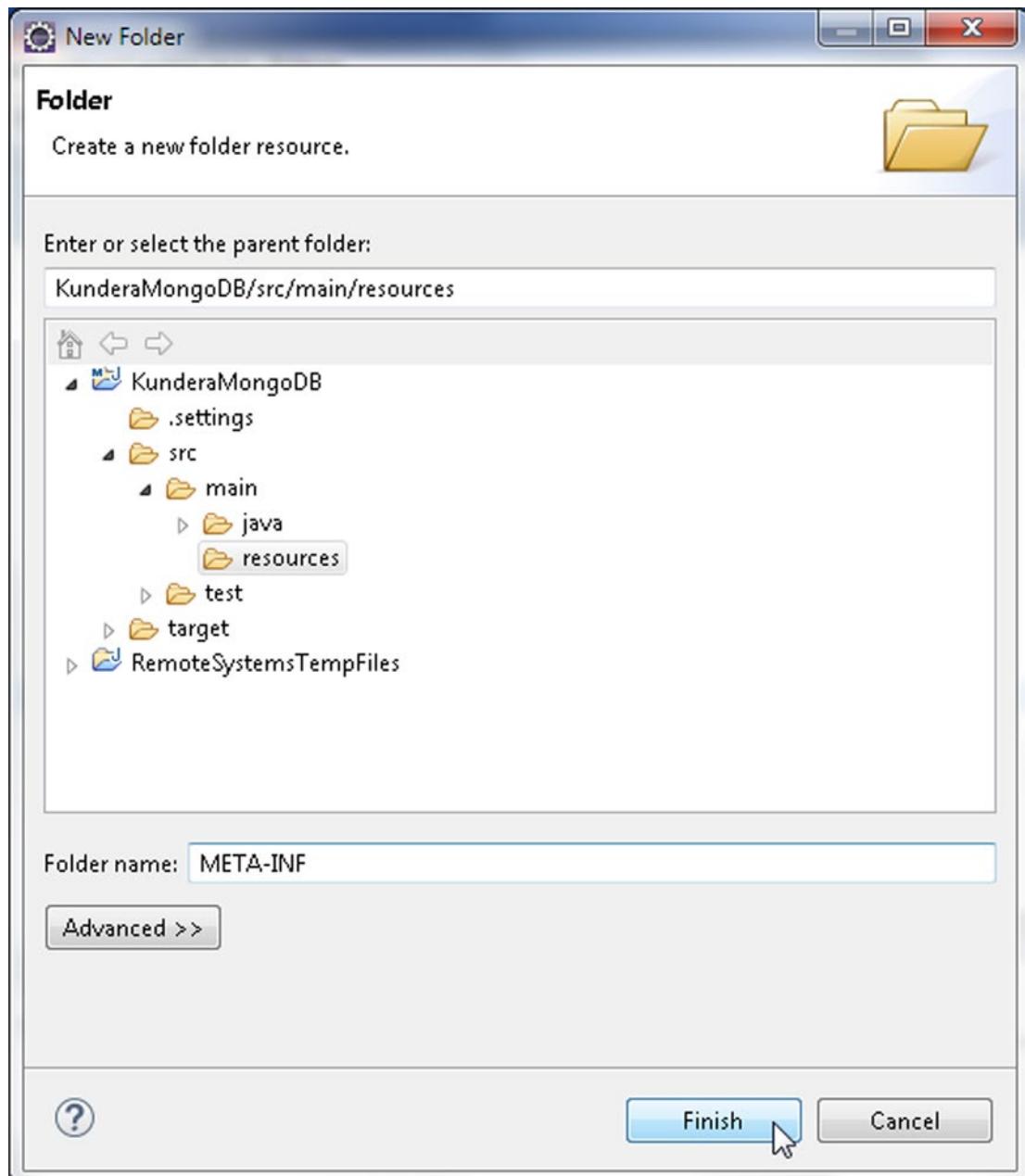


Figure 9-10. Creating the META-INF Folder

The META-INF folder gets created. Next, add a persistence.xml file to the META-INF folder.

1. To create the persistence.xml file select File > New > Other.
2. In the New window, select XML > XML File and click on Next as shown in Figure 9-11.

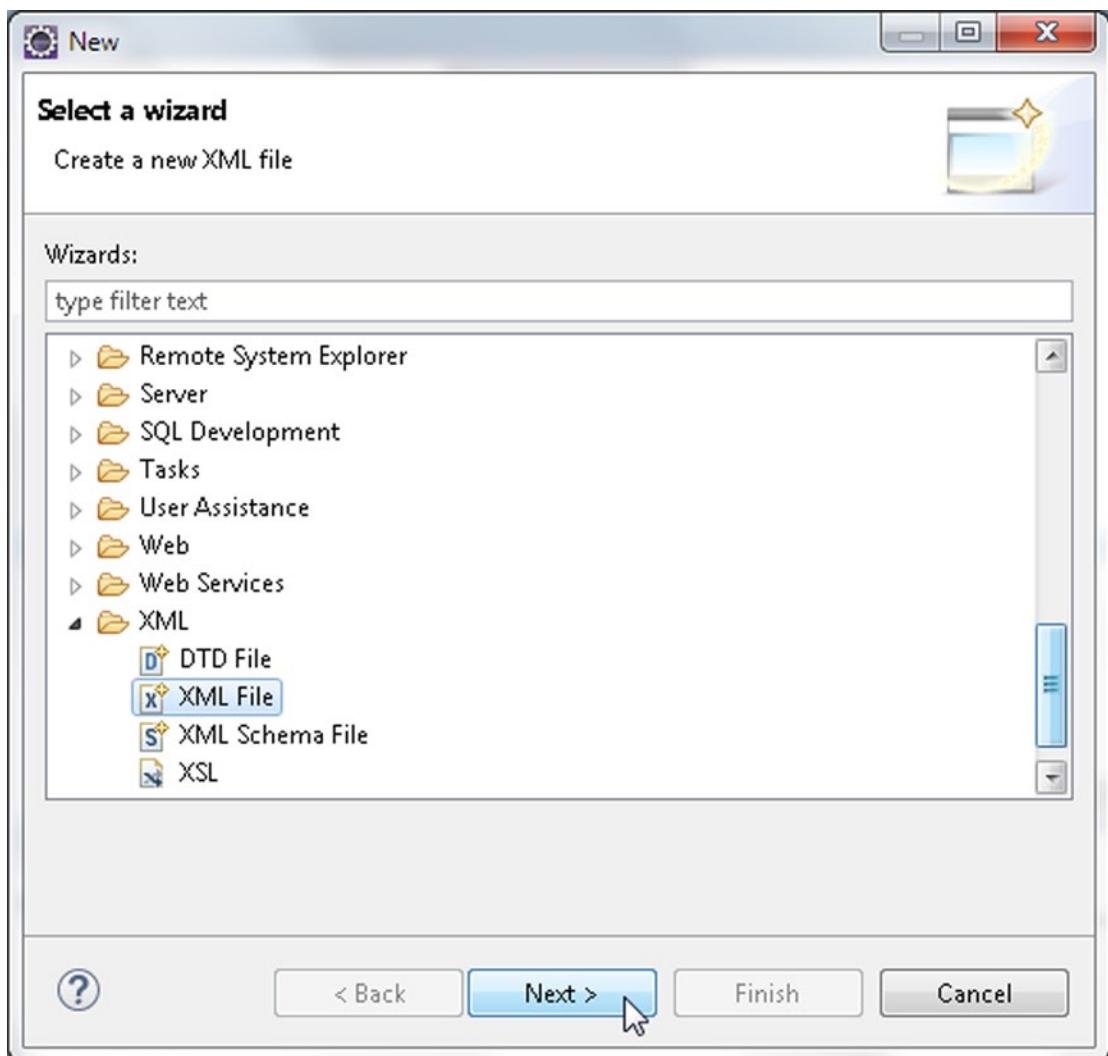


Figure 9-11. Creating an XML File

- In the New XML File wizard select the META-INF folder and specify File name as persistence.xml as shown in Figure 9-12. Click on Finish.

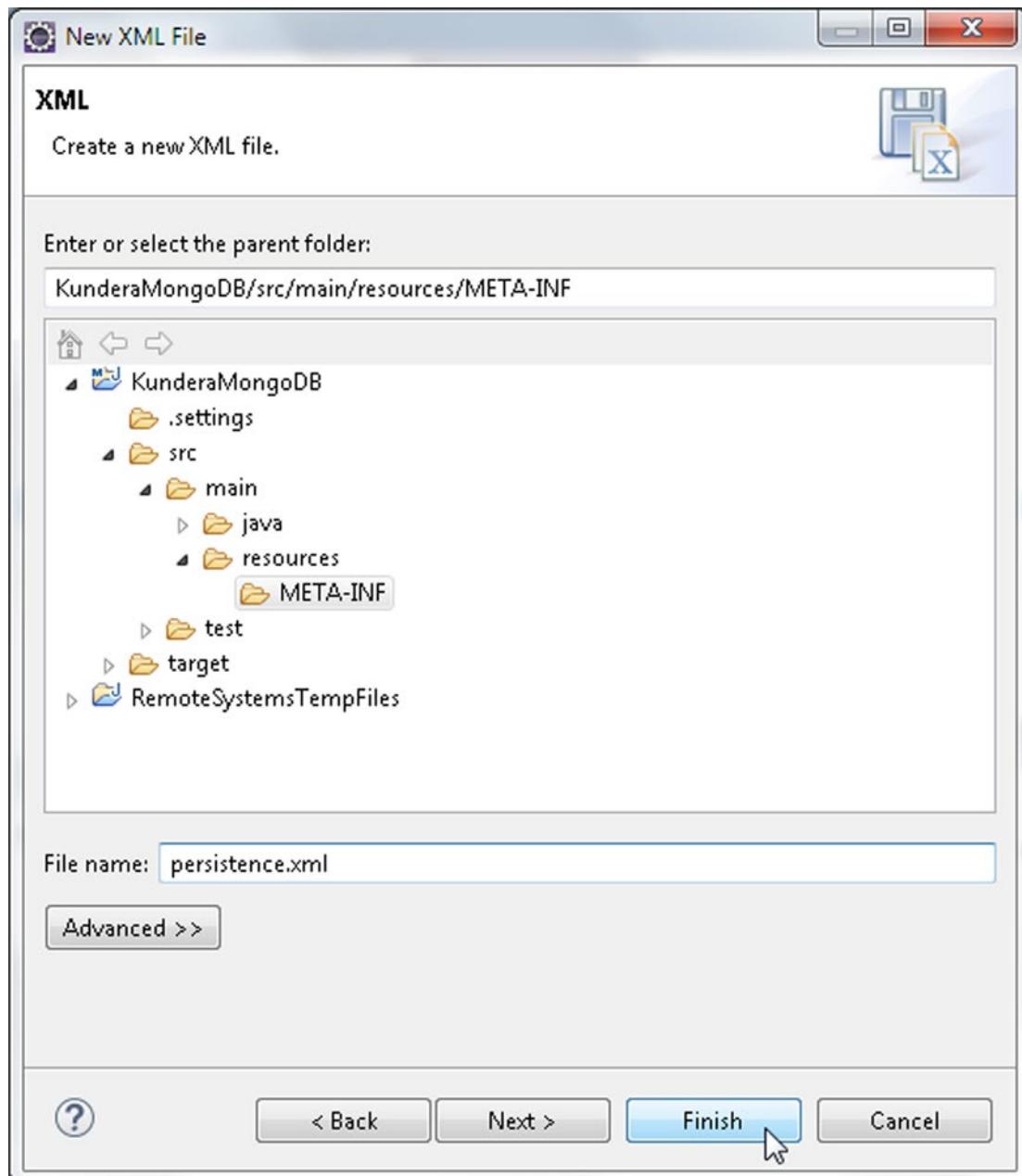


Figure 9-12. Creating the persistence.xml

The `persistence.xml` file gets added to the META-INF folder.

4. Copy the `persistence.xml` file listed earlier to the `persistence.xml` file in the Maven project as shown in Figure 9-13.

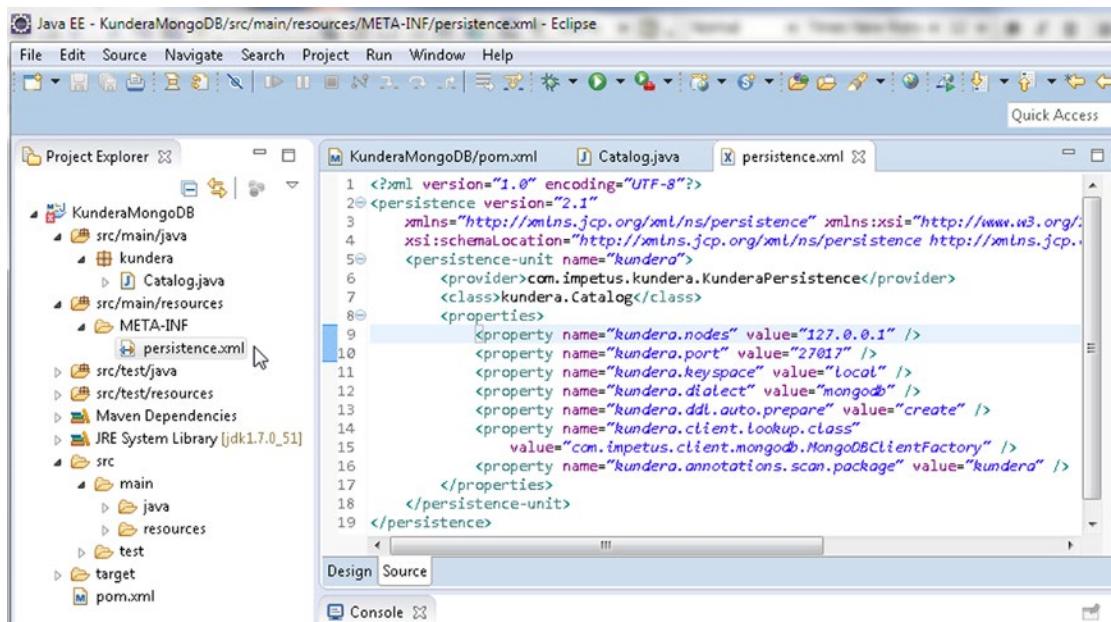


Figure 9-13. Persistence Configuration File `persistence.xml`

Creating a JPA Client Class

We have configured a JPA project for object/relational mapping to MongoDB database. Next, we shall run some CRUD operations using the JPA API. But, first we need to create a client class for the CRUD operations. We shall use a Java class as a client class.

1. Select **File > New > Other**.
2. In the **New** window, select **Java > Class** and click on **Next**.
3. In **New Java Class** wizard specify a Package (`kundera`) and a class Name (`KunderaClient`) as shown in Figure 9-14. Select the method stub for the `main` method to add to the class and click on **Finish**.

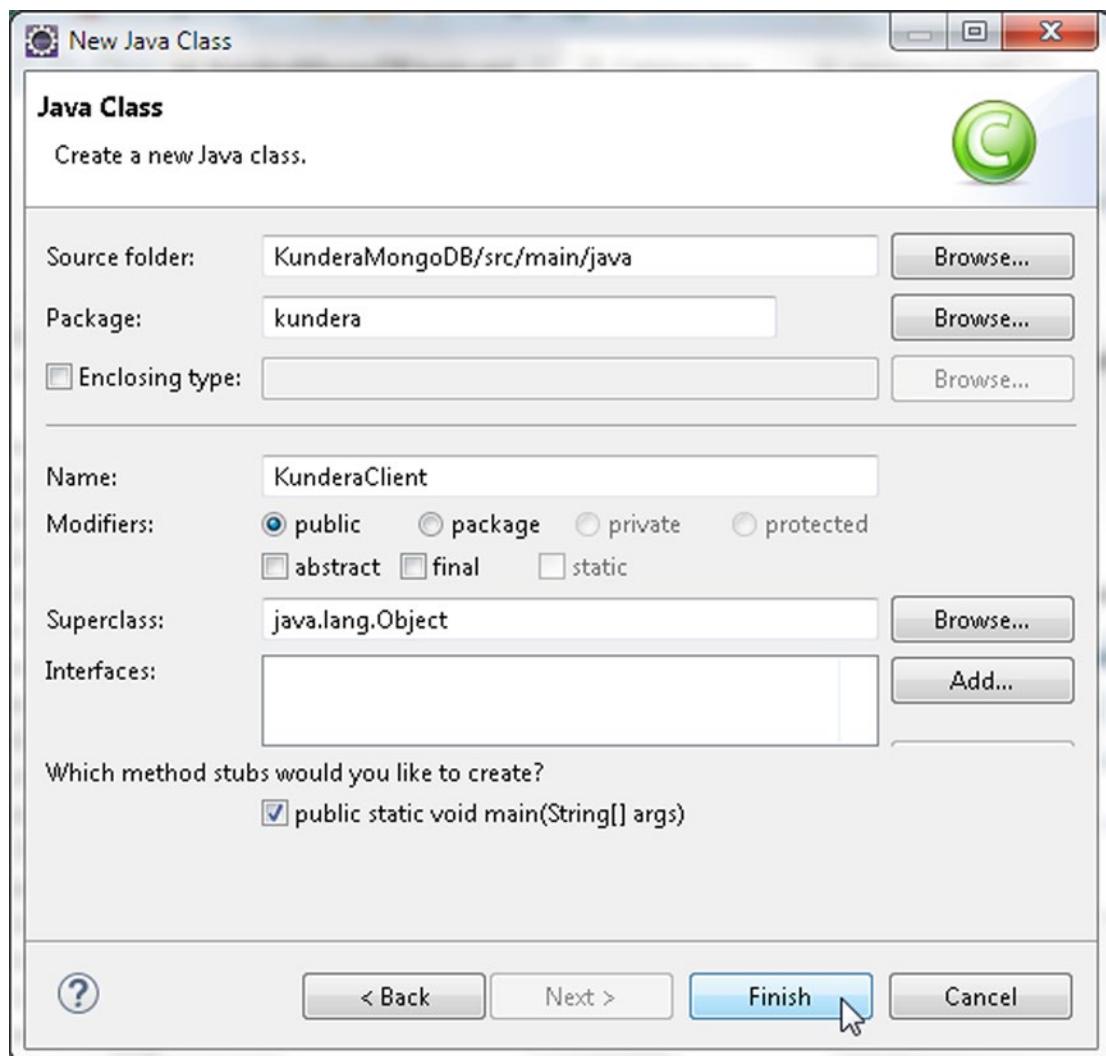


Figure 9-14. Creating the Client Class `KunderaClient`

The `kundera.KunderaClient` class gets added to the KunderaMongoDB Maven project as shown in Figure 9-15.

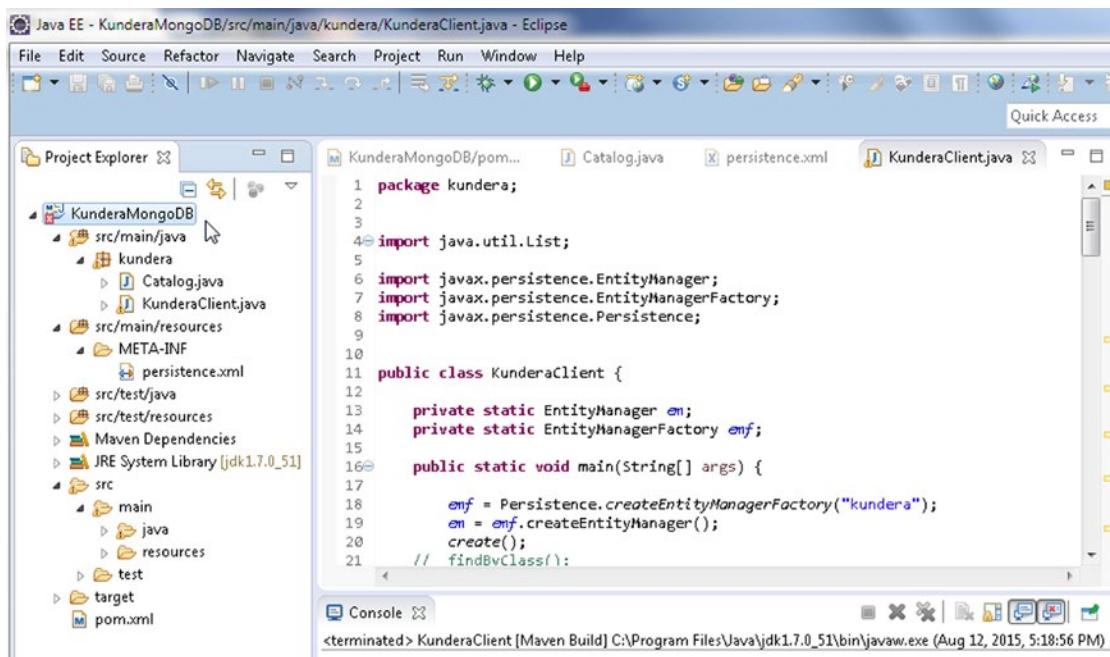


Figure 9-15. Maven Project Directory Structure including KunderaClient Class

We shall add the methods shown in Table 9-4 to the KunderaClient application to run CRUD operations on MongoDB. We shall invoke the methods from the `main` method and comment out all the methods to begin with. We shall uncomment one method at a time and run the KunderaClient application.

Table 9-4. KunderaClient Application Methods

Property	Description
<code>create()</code>	Create entities.
<code>findByClass()</code>	Find an entity using the entity class.
<code>query()</code>	Query the entities.
<code>update()</code>	Update entities.
<code>delete()</code>	Delete the entities.

Running CRUD Operations

In the next few subsections we shall create a catalog in the `catalog` collection and we shall add data to the `catalog`, find a catalog entry, update a catalog entry, and delete a catalog entry.

Creating a Catalog

In this section we shall add some documents to the catalog collection in MongoDB.

1. Add a method called `create()` to the `KunderaClient` class and invoke the method from the `main` method so that when the application is run the method gets invoked.

The JPA API is defined in the `javax.persistence` package. The `EntityManager` interface is used to interact with the persistence context. The `EntityManagerFactory` interface is used to interact with the entity manager factory for the persistence unit. The persistence class is used to obtain an `EntityManagerFactory` object in a Java SE environment.

2. Create a `EntityManagerFactory` object using Persistence class static method `createEntityManagerFactory(java.lang.String persistenceUnitName)`.
3. Create an `EntityManager` instance from the `EntityManagerFactory` object using the `createEntityManager()` method.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("kundera");
em = emf.createEntityManager();
```

4. In the `create()` method create an instance of the entity class `Catalog`. Using the set methods set the `catalogId`, `journal`, `publisher`, `edition`, `title`, and `author` fields.

```
Catalog catalog = new Catalog();
catalog.setCatalogId("catalog1");
catalog.setJournal("Oracle Magazine");
catalog.setPublisher("Oracle Publishing");
catalog.setEdition("November-December 2013");
catalog.setTitle("Engineering as a Service");
catalog.setAuthor("David A. Kelly");
```

5. Make the domain model instance managed and persistent using the `persist(java.lang.Object entity)` method in the `EntityManager` interface.

```
em.persist(catalog);
```

Similarly, other JPA instances may be persisted. When we run the `KunderaClient` class in a later section the `Catalog` entities get persisted to the MongoDB database.

Finding a Catalog Entry Using the Entity Class

The `EntityManager` class provides several methods for finding an entity instance. In this section we shall find a `Catalog` entity instance using the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method in which the first parameter is the entity class and the second parameter is the `_id` field value for the document to find.

1. Add a method called `findByClass()` to the `KunderaClient` class and invoke the method from the `main` method so that when the application is run the method gets invoked.
2. Invoke the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method using `Catalog.class` as the first arg and "catalog1" as the second arg.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
```

3. Invoke the get methods on the Catalog instance to output the entity fields.

```
System.out.println(catalog.getJournal());
System.out.println(catalog.getPublisher());
System.out.println(catalog.getEdition());
System.out.println(catalog.getTitle());
System.out.println(catalog.getAuthor());
```

The Catalog entity field values shall get output when the client class is run.

Finding a Catalog Entry Using a JPA Query

The Query interface is used to run a query in the Java Persistence query language and native SQL. The EntityManager interface provides several methods for creating a Query instance. In this section we shall run a Java Persistence query language statement by first creating an instance of Query using the EntityManager method `createQuery(java.lang.String qlString)` and subsequently invoking the `getResultSet()` method on the Query instance.

1. Add a method called `query()` to the KunderaClient class and invoke the method from the `main` method so that when the application is run the method gets invoked.
2. In the `query()` method invoke the `createQuery(java.lang.String qlString)` method to create a Query instance.
3. Supply the Java Persistence query language statement as `SELECT c FROM Catalog c`.

```
javax.persistence.Query query = em.createQuery("SELECT c FROM Catalog c");
```

4. Invoke the `getResultSet()` method on the Query instance to run the SELECT statement and return a `List<Catalog>` as result.

```
List<Catalog> results = query.getResultList();
```

5. Iterate over the List object using an enhanced for statement to output the fields of the Catalog instance.

```
for (Catalog catalog : results) {
    System.out.println(catalog.getCatalogId());
    System.out.println(catalog.getJournal());
    System.out.println(catalog.getPublisher());
    System.out.println(catalog.getEdition());
    System.out.println(catalog.getTitle());
    System.out.println(catalog.getAuthor());
}
```

The Catalog entity field values shall get output when the client class is run.

Updating a Catalog Entry

In this section we shall update a catalog entry using the Java Persistence API. The `persist()` method in `EntityManager` may be used to persist an updated entity instance.

1. Add a method called `update()` to the `KunderaClient` class and invoke the method from the `main` method so that when the application is run the method gets invoked. For example,
 - a. To update the `edition` column in document with primary key “catalog1” create an entity instance for the `catalog1` row using the `find(java.lang.Class<T>, java.lang.Object primaryKey)` method.
 - b. Subsequently set the `edition` field to the updated value using the `setEdition` method.
 - c. Persist the updated Catalog instance using the `persist(java.lang.Object entity)` method.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
catalog.setEdition("Nov-Dec 2013");
em.persist(catalog);
```

2. The Java Persistence query language provides the UPDATE clause to update a row. Create a `Query` instance using an UPDATE statement and the `createQuery(String)` method in `EntityManager`. Subsequently invoke the `executeUpdate()` method to execute the UPDATE statement.


```
em.createQuery("UPDATE Catalog c SET c.journal = 'Oracle-Magazine'").
executeUpdate();
```
3. The `journal` column in all the rows in the `catalog` column family gets updated. Having applied updates invokes the `query()` method to output the updated field values.

When the `KunderaClient` class is run the updated field `journal` should have the updated value `Oracle-Magazine` instead of `Oracle Magazine`.

Deleting a Catalog Entry

In this section we shall remove documents persisted in MongoDB using the Java Persistence API. The `remove(java.lang.Object entity)` method in `EntityManager` may be used to remove an entity instance.

1. Add a method called `delete()` to the `KunderaClient` class and invoke the method from the `main` method so that when the application is run the method gets invoked.

2. To remove the document with primary key “catalog1” create an entity instance for the catalog1 row using the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method. Subsequently invoke the `remove(java.lang.Object entity)` method to remove the document with primary key catalog1 from MongoDB.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
em.remove(catalog);
```

3. The Java Persistence query language provides the DELETE clause to delete a document. Create a Query instance using an DELETE statement and the `createQuery(String)` method in EntityManager. Subsequently invoke the `executeUpdate()` method to execute the DELETE statement.

```
em.createQuery("DELETE FROM Catalog c").executeUpdate();
```

All rows get deleted. The DELETE statement does not delete the document itself but deletes all the columns in the rows.

4. Having applied deletes, either using the `remove(java.lang.Object entity)` or using the DELETE Java Persistence query language statement invoke the `query()` method to output any Catalog instances persisted to catalog table.

When the client class is run no field values should get listed when the `query()` method is invoked subsequent to deleting the Catalog entries.

The Kundera-Mongo JPA Client Class

In the following subsections we shall install the Maven project and generate Eclipse IDE files for use with the Maven project. Subsequently we shall run the Kundera-Mongo JPA client class, which was discussed in the preceding section “Running JPA CRUD operations,” to invoke the different methods of the client class. The KunderaClient class used in this chapter is listed:

```
package kundera;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class KunderaClient {

    private static EntityManager em;
    private static EntityManagerFactory emf;

    public static void main(String[] args) {
```

```

emf = Persistence.createEntityManagerFactory("kundera");
em = emf.createEntityManager();
create();
findByName();
query();
update();
delete();
close();
}

private static void create() {
    Catalog catalog = new Catalog();
    catalog.setCatalogId("catalog1");
    catalog.setJournal("Oracle Magazine");
    catalog.setPublisher("Oracle Publishing");
    catalog.setEdition("November-December 2013");
    catalog.setTitle("Engineering as a Service");
    catalog.setAuthor("David A. Kelly");

    em.persist(catalog);

    catalog = new Catalog();
    catalog.setCatalogId("catalog2");
    catalog.setJournal("Oracle Magazine");
    catalog.setPublisher("Oracle Publishing");
    catalog.setEdition("November-December 2013");
    catalog.setTitle("Quintessential and Collaborative");
    catalog.setAuthor("Tom Haunert");

    em.persist(catalog);
}

private static void findByName() {
    Catalog catalog = em.find(Catalog.class, "catalog1");
    System.out.println(catalog.getJournal());
    System.out.println("\n");
    System.out.println(catalog.getPublisher());
    System.out.println("\n");
    System.out.println(catalog.getEdition());
    System.out.println("\n");
    System.out.println(catalog.getTitle());
    System.out.println("\n");
    System.out.println(catalog.getAuthor());
}

private static void query() {

    javax.persistence.Query query = em
        .createQuery("SELECT c FROM Catalog c");
    List<Catalog> results = query.getResultList();
}

```

```

        for (Catalog catalog : results) {
            System.out.println(catalog.getCatalogId());
            System.out.println("\n");
            System.out.println(catalog.getJournal());
            System.out.println("\n");
            System.out.println(catalog.getPublisher());
            System.out.println("\n");
            System.out.println(catalog.getEdition());
            System.out.println("\n");
            System.out.println(catalog.getTitle());
            System.out.println("\n");
            System.out.println(catalog.getAuthor());
        }
    }
    private static void update() {
        Catalog catalog = em.find(Catalog.class, "catalog1");
        catalog.setEdition("Nov-Dec 2013");
        em.persist(catalog);
        em.createQuery("UPDATE Catalog c SET c.journal = 'Oracle-Magazine'")
            .executeUpdate();
        System.out.println("After updating");
        System.out.println("\n");
        query();
    }
    private static void delete() {
        Catalog catalog = em.find(Catalog.class, "catalog1");
        em.remove(catalog);
        System.out.println("After removing catalog1");
        query();

        em.createQuery("DELETE FROM Catalog c").executeUpdate();
        System.out.println("\n");
        System.out.println("After removing all catalog entries");
        query();
    }
    private static void close() {
        em.close();
        emf.close();
    }
}

```

Installing the Maven Project

In the preceding section we developed the source code for a Maven project. In this section we shall build the Maven project, including generating the dependency JAR KunderaMongoDB-1.0.0.jar for the Maven project.

Right-click on pom.xml in Project Explorer and select Run As ▶ Maven install as shown in Figure 9-16 to build and install KunderaMongoDB-1.0.0.jar in the local repository in the target subdirectory.

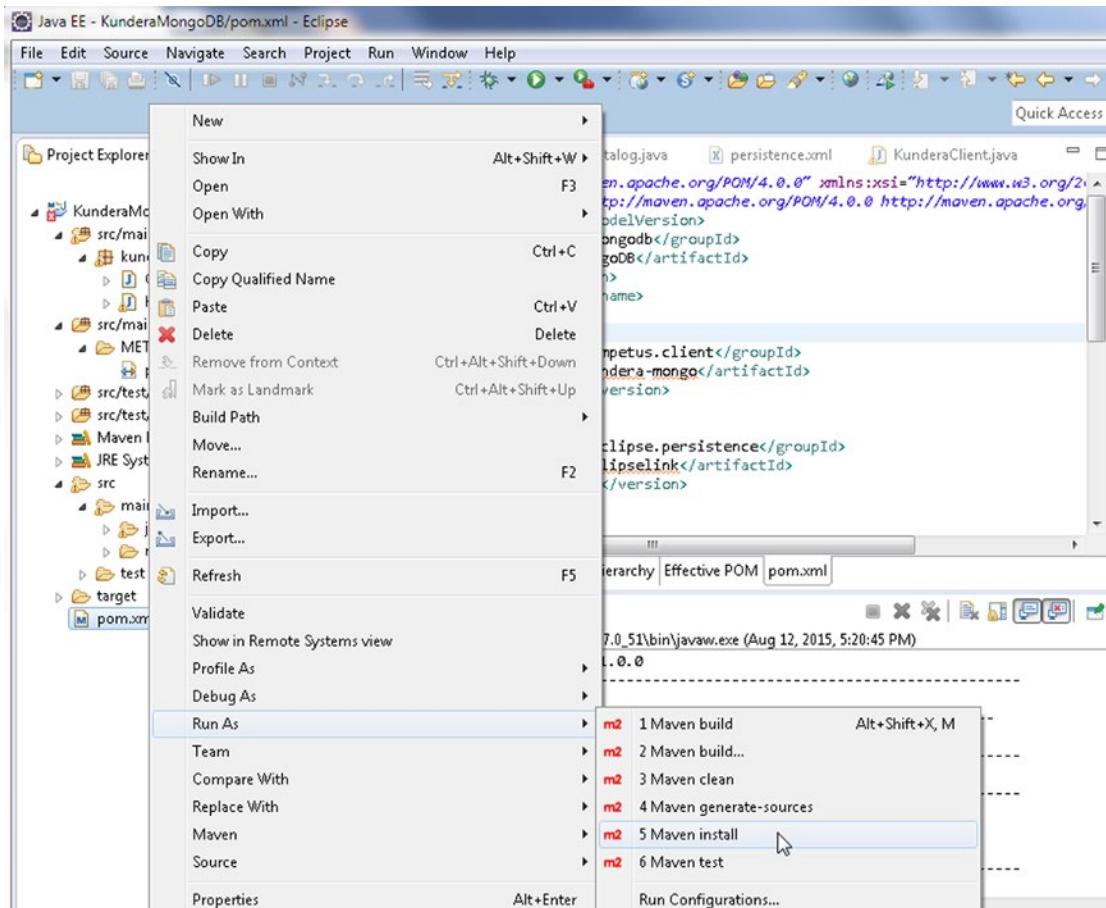


Figure 9-16. Installing Maven Project

Maven scans for projects and builds and installs the KunderaMongo project as shown in Figure 9-17. The output from the Maven install should include the message “BUILD SUCCESS.” If not, some error has occurred and the Maven project has not been installed. The /root/workspace/KunderaMongoDB/target/KunderaMongoDB-1.0.0.jar gets built and installed.

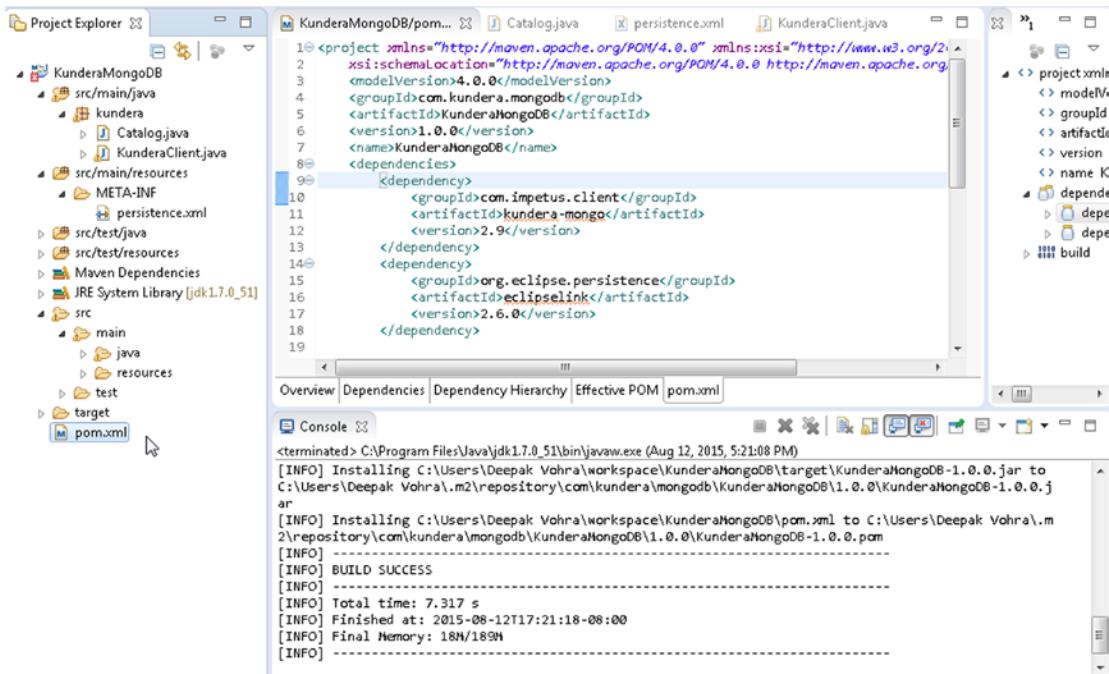


Figure 9-17. Output from Installing Maven Project

Next, we shall use the Maven Eclipse plugin to generate Eclipse IDE files for use with the Maven project. We shall run the following goals, listed in Table 9-5, from the Maven Eclipse plug-in.

Table 9-5. Eclipse Specific Configuration Properties

Goal	Description
eclipse:clean	Deletes the files used by Eclipse IDE.
eclipse:eclipse	Generates the Eclipse configuration files.

We need to create a run configuration for the `eclipse:clean` `eclipse:eclipse` goals.

1. Right-click on `pom.xml` and select Run As ➤ Run Configuration as shown in Figure 9-18.

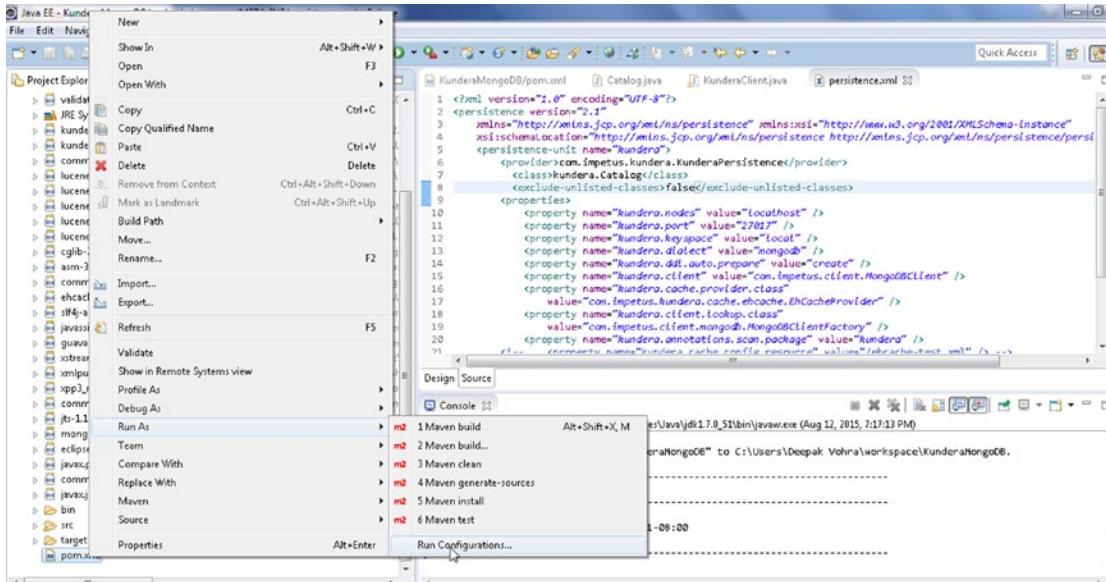


Figure 9-18. Selecting `pom.xml` ➤ `Run As` ➤ `Run Configuration`

2. Right-click on Maven Build and select New.
3. For the new run configuration, specify the following values and click on Apply.
 - Name: Eclipse (for example)
 - Base directory: The KunderaMongoDB project base directory
 - Goals: `eclipse:clean` `eclipse:eclipse`
4. Subsequently click on Run as shown in Figure 9-19.

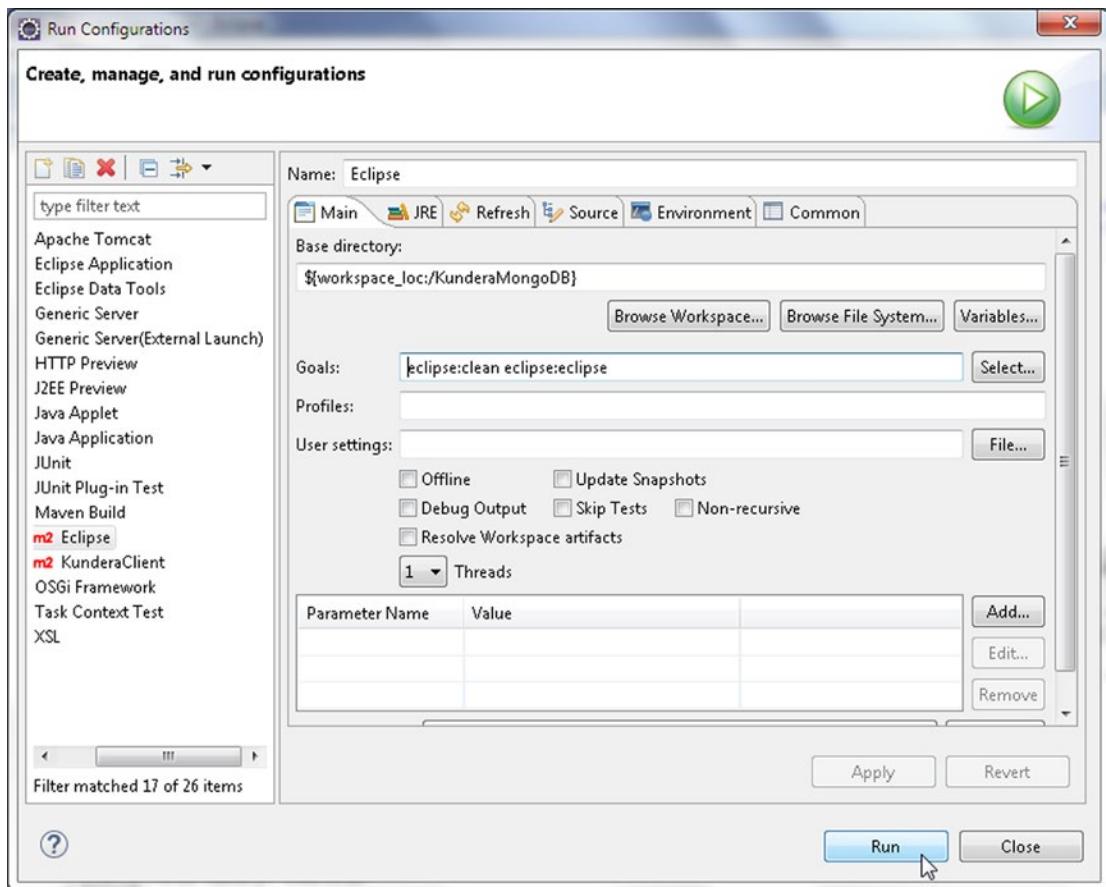


Figure 9-19. Configuring a New Run Configuration

If the Maven goals do not generate an error, a BUILD SUCCESS message should get output as shown in Figure 9-20.

```
<terminated> Eclipse [Maven Build] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:47:54 PM)
[INFO] Wrote settings to C:\Users\Deepak Vohra\workspace\KunderaMongoDB\.settings\org.eclipse.jdt.core.prefs
[INFO] Wrote Eclipse project for "KunderaMongoDB" to C:\Users\Deepak Vohra\workspace\KunderaMongoDB.
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.041 s
[INFO] Finished at: 2015-08-12T18:48:05-08:00
[INFO] Final Memory: 14M/189M
[INFO] -----
```

Figure 9-20. Output from the Maven Goals

Running the Kundera-Mongo JPA Client Class

In this section we shall run the `KunderaClient` class in the `KunderaMongoDB` project. We shall use the Maven `exec:java` goal from the Exec Maven plug-in to run the `KunderaClient` class. We specified the class to run using the `mainClass` parameter to the `exec-maven-plugin` configuration in `pom.xml`.

```
<configuration>
  <mainClass>kundera.KunderaClient</mainClass>
</configuration>
```

We shall run the `exec:java` goal in Eclipse to run the `KunderaClient` class. But, first we need to create a new run configuration in Eclipse.

1. Right-click on Maven Build in Run Configurations and New.
2. Specify a Name (`KunderaClient`, for example) and a goal (`exec:java`) and click on Apply and subsequently on Run as shown in Figure 9-21.

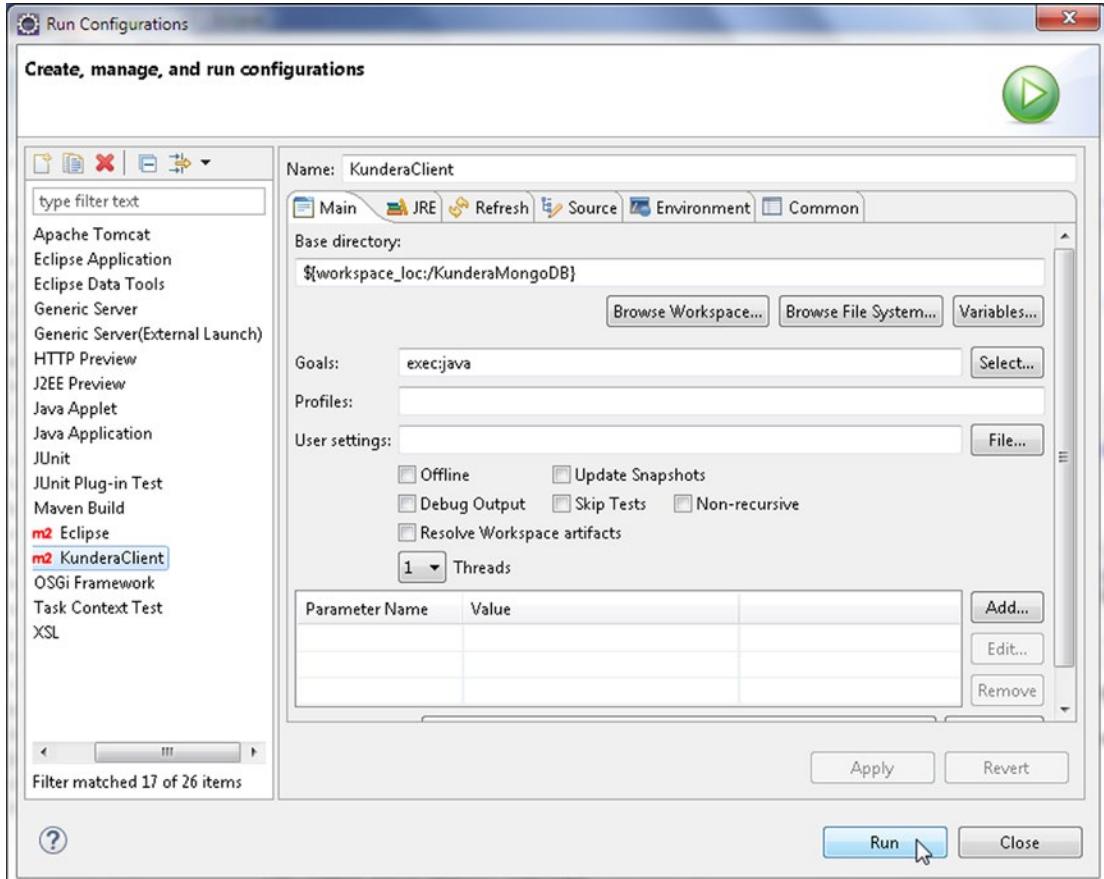


Figure 9-21. Running the `exec:java` Goal

The KunderaClient application gets run and the output for the invoked methods gets generated such as finding and listing entities. In the next section we shall invoke the KunderaClient class methods to create, find, update, and delete entities.

Invoking the KunderaClient Methods

Now it is time to invoke the methods.

1. First, invoke the `create()` method in the `main` method and comment out the other methods. When the Run Configuration for `exec:java` is run the KunderaClient application runs and the `create()` method gets invoked to create some entities.
2. Subsequently, run the `db.catalog.find()` method in Mongo shell to list the two entities added to MongoDB as shown in Figure 9-22.

```
> db.catalog.find()
{ "_id" : "catalog1", "publisher" : "Oracle Publishing", "journal" : "Oracle Magazine", "author" : "David A. Kelly", "edition" : "November-December 2013", "title" : "Engineering as a Service" }
{ "_id" : "catalog2", "publisher" : "Oracle Publishing", "journal" : "Oracle Magazine", "author" : "Tom Haunert", "edition" : "November-December 2013", "title" : "Quintessential and Collaborative" }
> _
```

Figure 9-22. Listing the Two Documents Added to MongoDB in Mongo Shell

3. Next, invoke the `findByClass()` method in the `main` method by uncommenting the method invocation.
4. Run the `exec:java` goal run configuration again. The `catalog1` entity fields get listed as shown in Figure 9-23.

The screenshot shows an IDE interface with several tabs at the top: KunderaMongoDB/pom.xml, Catalog.java, KunderaClient.java, and persistence.xml. Below these tabs is a code editor containing Java code. The code includes imports for EntityManager and EntityTransaction, and defines a Catalog class with methods like persist and findByClass. The findByClass method uses EntityManager to find a Catalog object by its class name ('catalog1') and prints its journal, publisher, and edition to System.out.

Console

```

<terminated> KunderaClient [Maven Build] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 7:09:22 PM)
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Oracle Magazine

Oracle Publishing

November-December 2013

Engineering as a Service

David A. Kelly
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.217 s
[INFO] Finished at: 2015-08-12T19:09:28-08:00
[INFO] Final Memory: 10M/189M
[INFO] -----

```

Figure 9-23. Output from Running the exec:java run Configuration for Invoking the `findByClass()` Method

5. Next, invoke the `query()` method using the `exec: java run` configuration. The output from running the Maven project run configuration is shown in Figure 9-24.

```
20 query();
```

```
<terminated> KunderaClient [Maven Build] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 7:10:42 PM)
```

```
catalog1
```

```
Oracle Magazine
```

```
Oracle Publishing
```

```
November-December 2013
```

```
Engineering as a Service
```

```
David A. Kelly
```

```
catalog2
```

```
Oracle Magazine
```

```
Oracle Publishing
```

```
November-December 2013
```

```
Quintessential and Collaborative
```

```
Tom Haunert
```

Figure 9-24. Output from Running exec:java run Configuration for Invoking the query() Method

6. Next, invoke the update() method to update the journal field.
7. Run the exec:java goal run configuration. The journal field gets updated and the updated field value get output as shown in Figure 9-25.

```
<terminated> KunderaClient [Maven Build] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 7:11:51 PM)
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
After updating

catalog1

Oracle-Magazine

Oracle Publishing

November-December 2013

Engineering as a Service

David A. Kelly
catalog2

Oracle-Magazine

Oracle Publishing

November-December 2013

Quintessential and Collaborative

Tom Haunert
```

Figure 9-25. Output from Running exec:java run Configuration for Invoking the update() Method

8. Next, invoke the delete() method and run the KunderaClient application using the run configuration for exec:java. As the output in Figure 9-26 indicates, the entities get listed after removing one and after removing both entities.

The screenshot shows an IDE interface with several tabs at the top: KunderaMongoDB/pom.xml, Catalog.java, KunderaClient.java (selected), and persistence.xml. The KunderaClient.java tab contains Java code for managing EntityManagers:

```
1 package kundera;
2
3 import java.util.List;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
```

The Console tab shows the following output:

```
<terminated> KunderaClient [Maven Build] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 7:13:22 PM)
After removing catalog1
catalog2

Oracle Magazine

Oracle Publishing

November-December 2013

Quintessential and Collaborative

Tom Haunert

After removing all catalog entries
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.276 s
[INFO] Finished at: 2015-08-12T19:13:30-08:00
[INFO] Final Memory: 10M/189M
[INFO] -----
```

Figure 9-26. Output from Running exec:java run Configuration for Invoking the delete() Method

Summary

In this chapter we used Kundera with the kundera-mongo module to perform CRUD operations in MongoDB using a Java client class with a JPA entity class, and a `persistence.xml` configuration file. The Kundera-Mongo application was developed as a Maven project in Eclipse, and built and run using goals from the Maven Eclipse plug-in and Exec Maven plug-in. In the next chapter we shall use Spring Data with MongoDB.

CHAPTER 10



Using Spring Data with MongoDB

Spring Data is designed for new data access technologies such as non-relational databases. MongoDB is a non-relational NoSQL database with benefits such as scalability, flexibility, and high performance. The Spring Data MongoDB project adds Spring Data functionality to the MongoDB server. This chapter explains how to use the Spring Data MongoDB project to access MongoDB and perform CRUD operations on the database in Eclipse. We shall use Maven as the build automation tool. The chapter includes the following topics:

- Setting up the environment
- Creating a Maven project
- Installing Spring Data MongoDB
- Configuring JavaConfig
- Creating a model
- Using Spring Data with MongoDB with Template
- Using Spring Data repositories with MongoDB

Setting Up the Environment

We need to download and install the following software for this chapter.

- Eclipse IDE for Java EE Developers. Download from www.eclipse.org/downloads. Eclipse 4.4 Luna used in this chapter.
- MongoDB 3.0.5 (or later version) binary distribution from www.mongodb.org/downloads.
- Java SE 7 from www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Double-click on the MongoDB binary distribution to install MongoDB. Add the bin directory (C:\Program Files\MongoDB\Server\3.0\bin) of the MongoDB installation to the PATH environment. Create a directory C:\data\db for the MongoDB data if not already created for an earlier chapter.

Start the MongoDB server with the following command.

```
>mongod
```

Creating a Maven Project

First, we need to create a Maven project in Eclipse.

1. Select File > New > Other.
2. In the New window, select the Maven > Maven Project wizard and click on Next as shown in Figure 10-1.

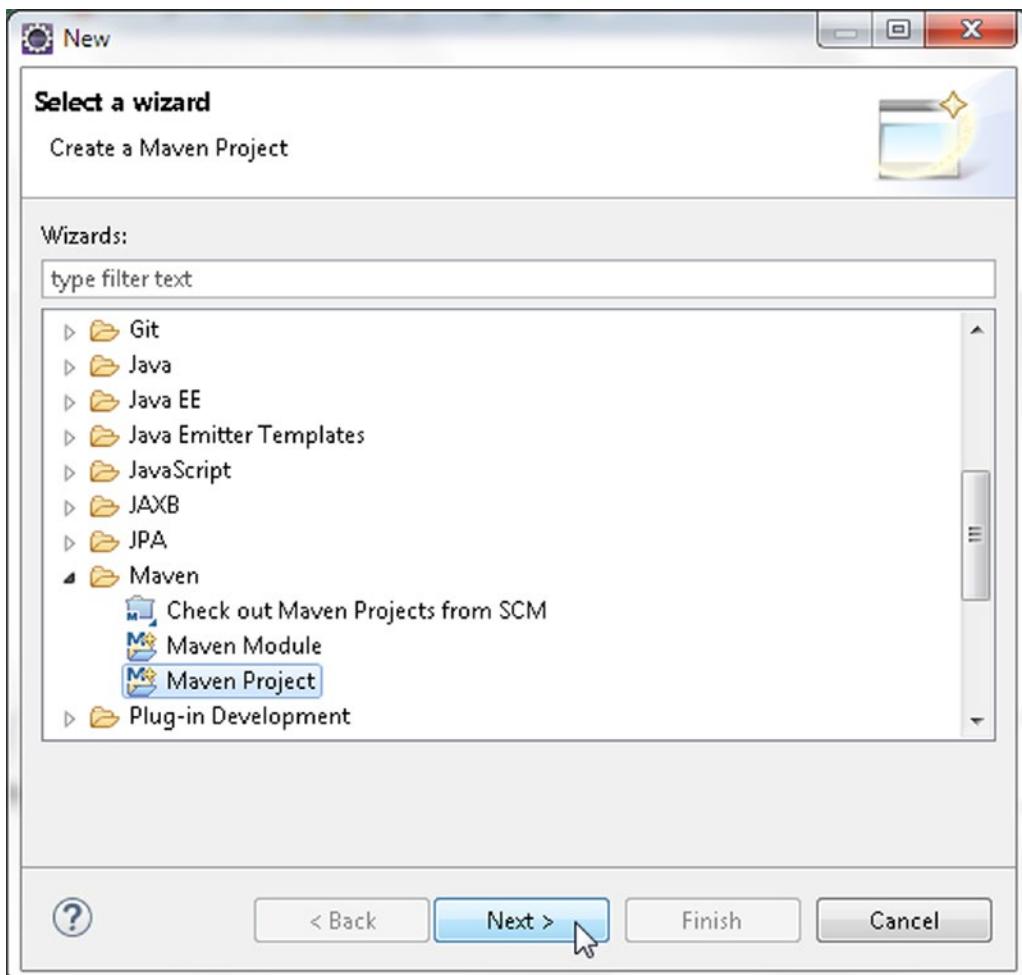


Figure 10-1. Creating a New Maven Project

3. The New Maven Project wizard gets started. Select the Create a simple project check box and the Use default Workspace location check box as shown in Figure 10-2. Then click on Next.

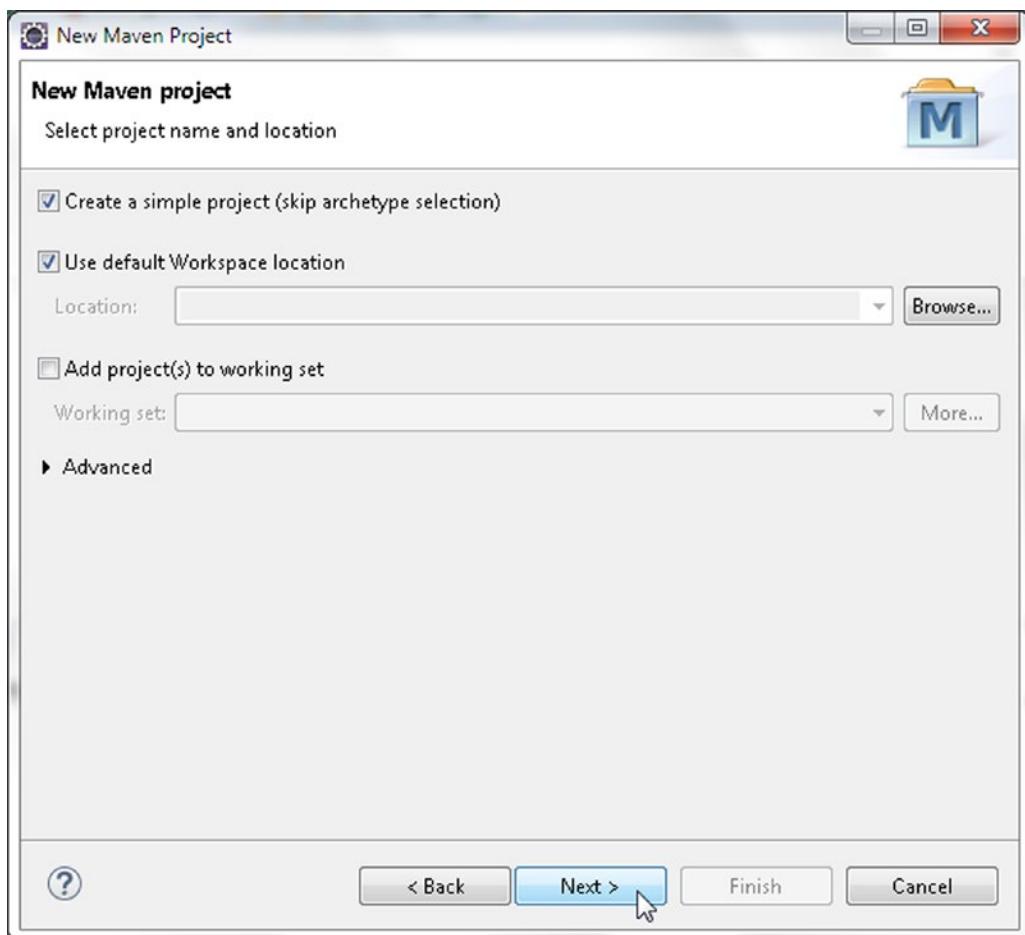


Figure 10-2. The New Maven Project Wizard

4. In Configure project specify the following settings and click on Finish as shown in Figure 10-3.

- Group Id: com.spring.mongodb
- Artifact Id: SpringDataMongo
- Version: 1.0.0
- Packaging: jar
- Name: SpringDataMongo

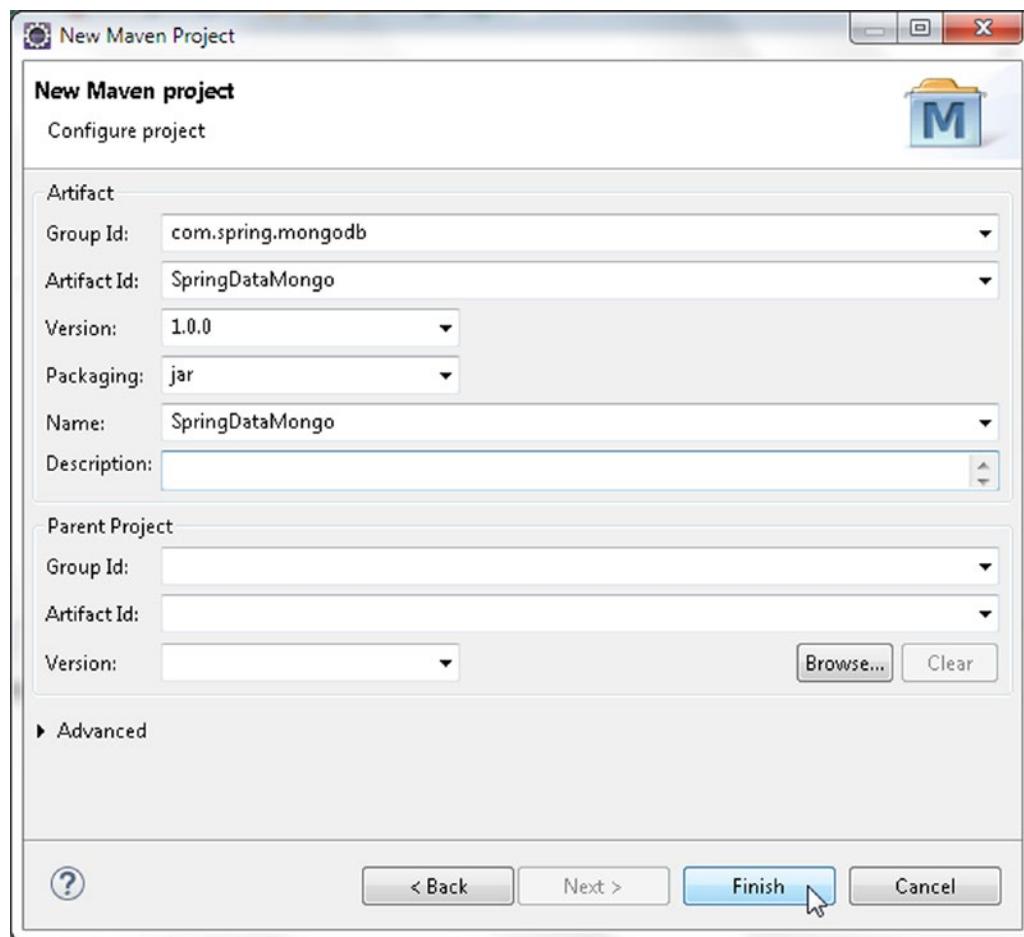


Figure 10-3. Configuring a New Maven Project

A Maven project (SpringDataMongo) gets created as shown in Package Explorer as shown in Figure 10-4.

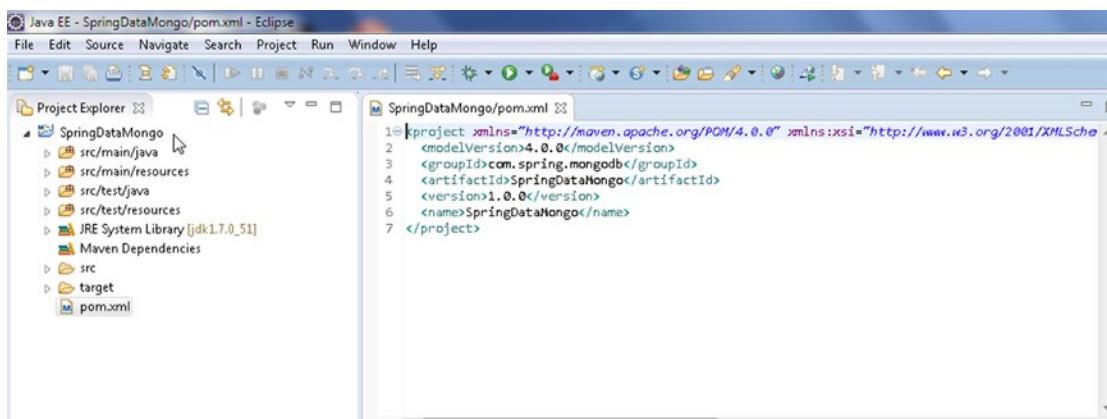


Figure 10-4. The New Maven Project SpringDataMongo

The Java Build Path for the project should include the Maven dependencies including the Spring Data MongoDB project dependency. Installing Spring Data MongoDB and other dependencies is discussed in the next section. We need to add some classes to the Maven project to use Spring Data with MongoDB. Add Java Classes listed in Table 10-1.

Table 10-1. Java Classes

Class	Description
com.mongo.config.SpringMongoApplicationConfig	JavaConfig class.
com.mongo.core.App	Java application for using Spring Data with MongoDB with Template.
com.mongo.model.Catalog	Model class.
com.mongo.repositories.CatalogRepository	Implementation class for MongoDB specific repository.
com.mongo.service.CatalogService	Service class to invoke CRUD operations on MongoDB Repository.

The Java classes in the Maven project are shown in Figure 10-5.

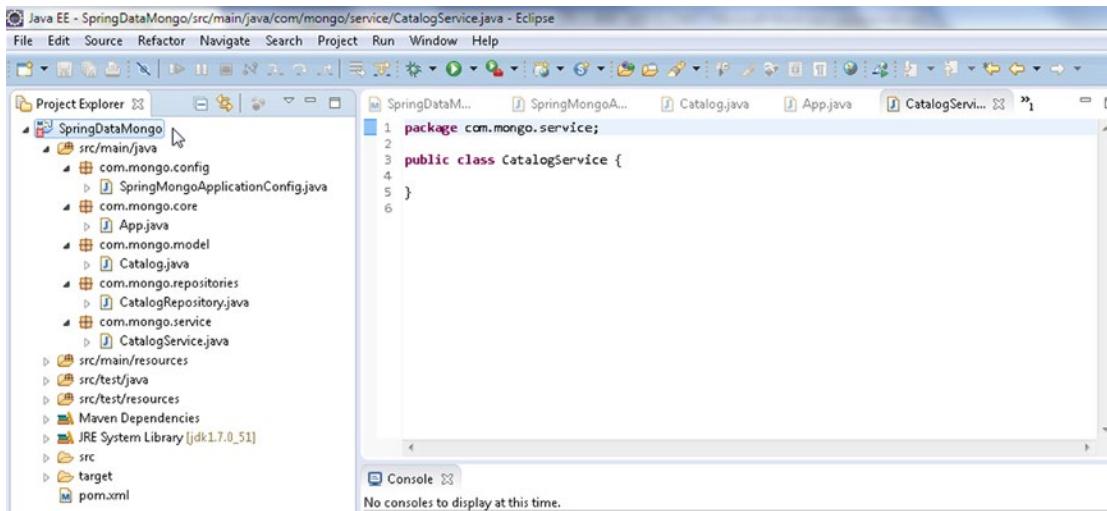


Figure 10-5. Java Classes in Maven Project

In subsequent sections we shall install and use the Spring Data MongoDB project. Unless noted otherwise, before running an application, `App.java` or `CatalogService.java`, drop the catalog collection in the local database if it already exists using the method `db.catalog.drop()` in Mongo shell.

```
>use local
>db.catalog.drop()
```

Installing Spring Data MongoDB

The Maven project includes a `pom.xml` in the root directory of the Maven project to specify the dependencies for the project and the build configuration for the project. Specify the dependency(ies) listed in Table 10-2 in `pom.xml`.

Table 10-2. Maven Project Dependencies

Dependency Group Id	Artifact Id	Version	Description
org.springframework.data	spring-data-mongodb	1.7.2.RELEASE	Spring Data MongoDB.

Specify the `maven-compiler-plugin` and `maven-eclipse-plugin` plug-in in the build configuration. The `pom.xml` to use the Spring Data MongoDB project is listed.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.spring.mongodb</groupId>
```

```

<artifactId>SpringDataMongo</artifactId>
<version>1.0.0</version>
<name>SpringDataMongo</name>
<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-mongodb</artifactId>
        <version>1.7.2.RELEASE</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.0</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-eclipse-plugin</artifactId>
            <version>2.9</version>
            <configuration>
                <downloadSources>true</downloadSources>
                <downloadJavadocs>true</downloadJavadocs>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>

```

Configuring JavaConfig

Configure the Spring environment with *plain old Java objects* (POJOs) using JavaConfig. A POJO is an ordinary Java object without any special constraints of Java object models or conventions. The base class for Spring Data MongoDB configuration with JavaConfig is `org.springframework.data.mongodb.config.AbstractMongoConfiguration`.

1. Create a class, `SpringMongoApplicationConfig`, which declares some `@Bean` methods and extends the `org.springframework.data.mongodb.config.AbstractMongoConfiguration` class.
2. Annotate the class with `@Configuration`, which indicates that the class is processed by the Spring container to generate bean definitions and service requests for the beans at runtime.

3. Declare a @Bean annotated method that returns a MongoClient instance. The SpringMongoApplicationConfig class must implement the inherited abstract methods getDatabaseName() and mongo(). The localhost host name (the IP address may also be used) and port number 27017 are used to create a MongoClient instance.
4. Also override the non-abstract method getMappingBasePackage to return the package (com.mongo.model) in which the model class is defined.

The Spring configuration class SpringMongoApplicationConfig is listed below.

```
package com.mongo.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.AbstractMongoConfiguration;
import org.springframework.context.annotation.Bean;
import com.mongo.service.CatalogService;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;

import java.util.Arrays;

@Configuration
public class SpringMongoApplicationConfig extends AbstractMongoConfiguration {

    @Override
    @Bean
    public Mongo mongo() throws Exception {
        return new MongoClient(Arrays.asList(new ServerAddress("localhost",
                27017)));
    }

    @Override
    protected String getDatabaseName() {
        return "local";
    }

    @Override
    protected String getMappingBasePackage() {
        return "com.mongo.model";
    }
}
```

Creating a Model

Next, create the model class to use with the Spring Data MongoDB project. A domain object to be persisted to MongoDB server must be annotated with `@Document`.

1. Create a POJO class `Catalog` in the `com.mongo.model` package.
2. Add fields for `id`, `journal`, `edition`, `publisher`, `title`, and `author` and the corresponding get/set methods.
3. Annotate the `id` field with `@Id`.
4. Add a constructor that may be used to construct a `Catalog` instance.

The `Catalog` entity is listed below.

```
package com.mongo.model;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
@Id
public class Catalog {
    private String id;
    private String journal;
    private String publisher;
    private String edition;
    private String title;
    private String author;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getJournal() {
        return journal;
    }

    public void setJournal(String journal) {
        this.journal = journal;
    }

    public String getPublisher() {
        return publisher;
    }

    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
}
```

```

public String getEdition() {
    return edition;
}

public void setEdition(String edition) {
    this.edition = edition;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public Catalog(String id, String journal, String publisher, String edition,
               String title, String author) {
    id = this.id;
    this.journal = journal;
    this.publisher = publisher;
    this.edition = edition;
    this.title = title;
    this.author = author;
}

}

```

Using Spring Data MongoDB with Template

In this section we shall use a MongoDB template to perform CRUD operations on MongoDB server. The term “template” refers to an implementation of MongoDB operations such as create, find, update, delete, aggregate, upsert, and count. The common CRUD operations on a MongoDB datastore may be performed using the `org.springframework.data.mongodb.core.MongoOperations` interface.

1. Create a Java class `com.mongo.core.App` to run CRUD operations on MongoDB server.

2. The `org.springframework.data.mongodb.core.MongoTemplate` class implements the `MongoOperations` interface. A `MongoTemplate` instance may be obtained using the `ApplicationContext`. Create an `ApplicationContext` as follows.

```
ApplicationContext context = new AnnotationConfigApplicationContext(
    SpringMongoApplicationConfig.class);
```

The `getBean(String name, Class requiredType)` method returns a named bean of the specified type. The bean name for a `MongoTemplate` is `mongoTemplate`. The class type is `MongoOperations.class`.

```
MongoOperations ops = context.getBean("mongoTemplate", MongoOperations.class);
```

3. The `MongoOperations` instance may be used to perform various CRUD operations on a domain object stored in the MongoDB. Add the static methods listed in Table 10-3 to the `App.java` class and add method invocations for the methods in the `main` method. The `App` class method names are same or similar to the `MongoOperations` method names.
4. Add class variables for a `MongoOperations` instance `ops` and two `Catalog` instances `catalog1` and `catalog2`.

```
static MongoOperations ops;
static Catalog catalog1;
static Catalog catalog2;
```

In the following subsections we shall invoke the methods listed in Table 10-3 to create a collection, create document instances, and run CRUD operations.

Table 10-3. Methods in `App.java`

Method	Description
<code>createCollection()</code>	Creates a collection.
<code>createCatalogInstances()</code>	Creates some Catalog instances.
<code>addDocument()</code>	Adds a Document.
<code>addDocumentBatch()</code>	Adds a document batch.
<code>findById()</code>	Finds a document by Id.
<code>findOne()</code>	Finds one document.
<code>findAll()</code>	Finds all documents.
<code>find()</code>	Finds documents.
<code>updateFirst()</code>	Updates the first document.
<code>updateMulti()</code>	Updates multiple documents.
<code>remove()</code>	Removes documents.

Creating a MongoDB Collection

First, we need to create a collection to which to add documents. The `MongoOperations` interface provides the overloaded method `createCollection()` to create a collection. Each of the `createCollection()` methods returns a `com.mongodb.DBCollection` instance as discussed in Table 10-4.

Table 10-4. Overloaded `createCollection()` Methods in `MongoOperations`

Method	Description
<code>createCollection(Class<T> entityClass)</code>	Creates an uncapped collection with a name based on the specified entity class.
<code>createCollection(Class<T> entityClass, CollectionOptions collectionOptions)</code>	Creates a collection with a name based on the specified entity class and using the specified collection options.
<code>createCollection(String collectionName)</code>	Creates an uncapped collection with the specified name.
<code>createCollection(String collectionName, CollectionOptions collectionOptions)</code>	Creates a collection with a name based on the specified entity class and using the specified collection options.

We shall create a collection in the `createCollection()` class method in the App application. First, find if a collection by the name `catalog`, which we want to create, already exists. The `MongoOperations` interface provides the `collectionExists(String collectionName)` and `collectionExists(Class<T> entityClass)` methods to find if a collection exists.

1. In an `if-else` statement find if the `catalog` collection exists. If the collection does not exist create the collection using the `createCollection(String collectionName)` method. If the collection does exist drop the collection using the `dropCollection(String collectionName)` method and create the collection again using the `createCollection(String collectionName)` method. The `createCollection()` class method is listed:

```
private static void createCollection() {
    if (!ops.collectionExists("catalog")) {
        ops.createCollection("catalog");
    } else {
        ops.dropCollection("catalog");
        ops.createCollection("catalog");
    }
}
```

2. Invoke the `createCollection()` method in the `main` method.
3. Run the `App.java` application to create a collection called `catalog` in the local database. To run `App.java` right-click on `App.java` in Package Explorer and select Run As ▶ Java Application as shown in Figure 10-6.

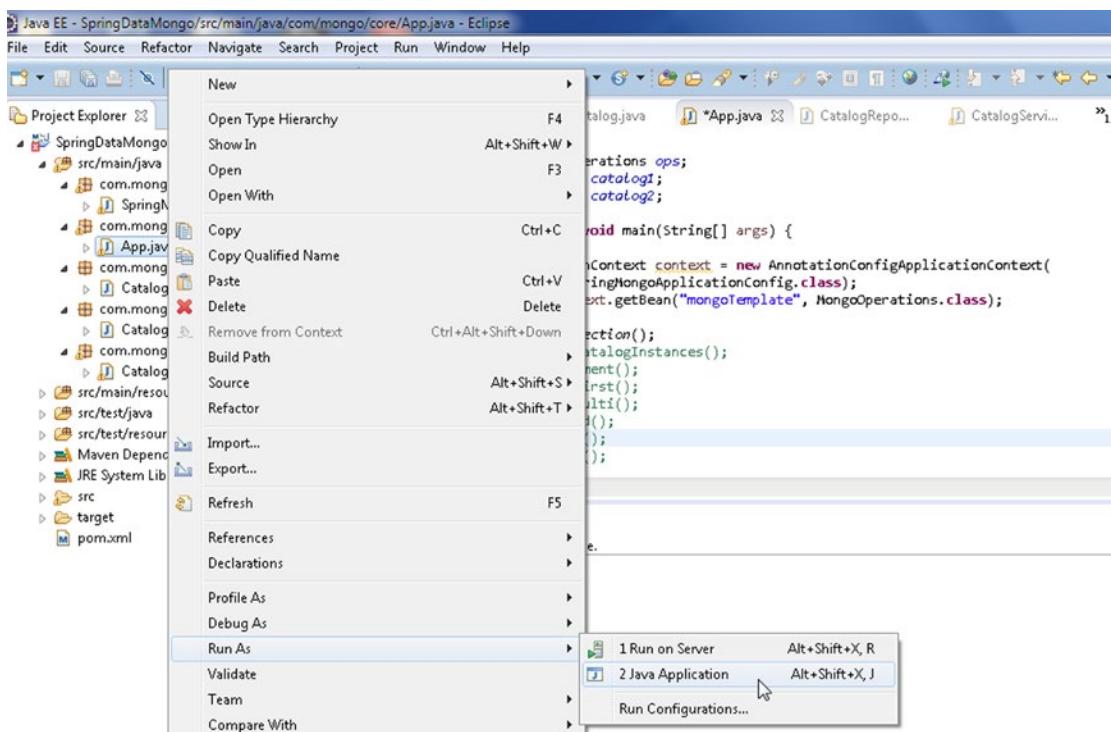


Figure 10-6. Running the App.java Application

4. Run the command `show collections` to list the collections after running the `App.java` application to create a collection. The `catalog` collection gets listed as shown in Figure 10-7.

```
C:\Users\Deepak Vohra>mongo
2015-08-12T05:53:23.742-0700 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
MongoDB shell version: 3.0.5
connecting to: test
> use local
switched to db local
> show collections
startup_log
system.indexes
> show collections
catalog
startup_log
system.indexes
>
```

Figure 10-7. Listing the catalog Collection in the Mongo Shell

Creating Document Instances

We shall be performing CRUD operations on MongoDB documents, and we have added class variables for the document instances so that we may use the same document instance for the different method invocations and do not have to create a new document instance in each method. In the `createCatalogInstances()` method, create two instances of `Catalog` using the class constructor. One or more of the field values may be kept empty as the only required field is `_id`, which is generated automatically.

```
catalog1 = new Catalog("catalog1", "Oracle Magazine", "Oracle Publishing",
"November-December 2013", "Engineering as a Service",
"David A. Kelly");
catalog2 = new Catalog("catalog2", "Oracle Magazine", "Oracle Publishing",
"November-December 2013", "Quintessential and Collaborative", "Tom Haunert");
```

Adding a Document

The `MongoOperations` interface provides the overloaded `save()` methods and the overloaded `insert()` methods to add documents. The `insert()` method is used to initially store a document in the database while the `save()` method is used to add a new document if a document with the same `_id` does not exist and update the document if the document with the same `_id` already exists. If it is not known if a document with a particular `_id` could already be in the database, use the `save()` method because the `insert()` method would fail if a document with the same `_id` already exists. We shall use the `save()` method in this section. In the next subsection we shall also discuss the `insert()` method to add a collection of documents.

The `save()` method updates the document if a document with the same `_id` is already in the database. If a document with the same `_id` does not exist a new document is added, and an upsert is performed. The `_id` is generated automatically. If the entity type of the object to save has an `Id` property, a property annotated with `@Id`, it is set with the generated `_id` value from MongoDB. If the `Id` property is of type `String` its value is set using an `ObjectId` instance created from the `_id` field value. The two overloaded `save()` methods are discussed in Table 10-5.

Table 10-5. Overloaded `save()` Methods

Method	Description
<code>save(Object objectToSave)</code>	Save the object to the collection for the entity type of the object to save.
<code>save(Object objectToSave, String collectionName)</code>	Save the object to the specified collection.

1. In the `addDocument()` method create an instance of `Catalog`.

```
catalog1 = new Catalog("catalog1", "Oracle Magazine", "Oracle Publishing",
"November-December 2013", "Engineering as a Service", "David A. Kelly");
```

2. Invoke the `save(Object objectToSave, String collectionName)` method using the `Catalog` instance as the first argument and collection name `catalog` as the second argument. A collection is created implicitly if not already in the database. For example, the `catalog` collection does not have to be in the MongoDB database before invoking the `save()` method with collection name `catalog`.

```
ops.save(catalog1, "catalog");
```

If the `save(Object objectToSave)` method is used to save to a particular collection and the collection does not already exist, a collection with the same name as the object class is created implicitly.

```
ops.save(catalog1);
```

3. Output the automatically generated `_id` that is set in the `Id` property.

```
System.out.println("MongoDB generated Id: " + catalog1.getId());
```

4. Similarly add another document.

```
catalog2 = new Catalog("catalog2", "Oracle Magazine", "Oracle Publishing",
    "November-December 2013", "Quintessential and Collaborative", "Tom Haunert");
ops.save(catalog2, "catalog");
System.out.println("MongoDB generated Id: " + catalog2.getId());
```

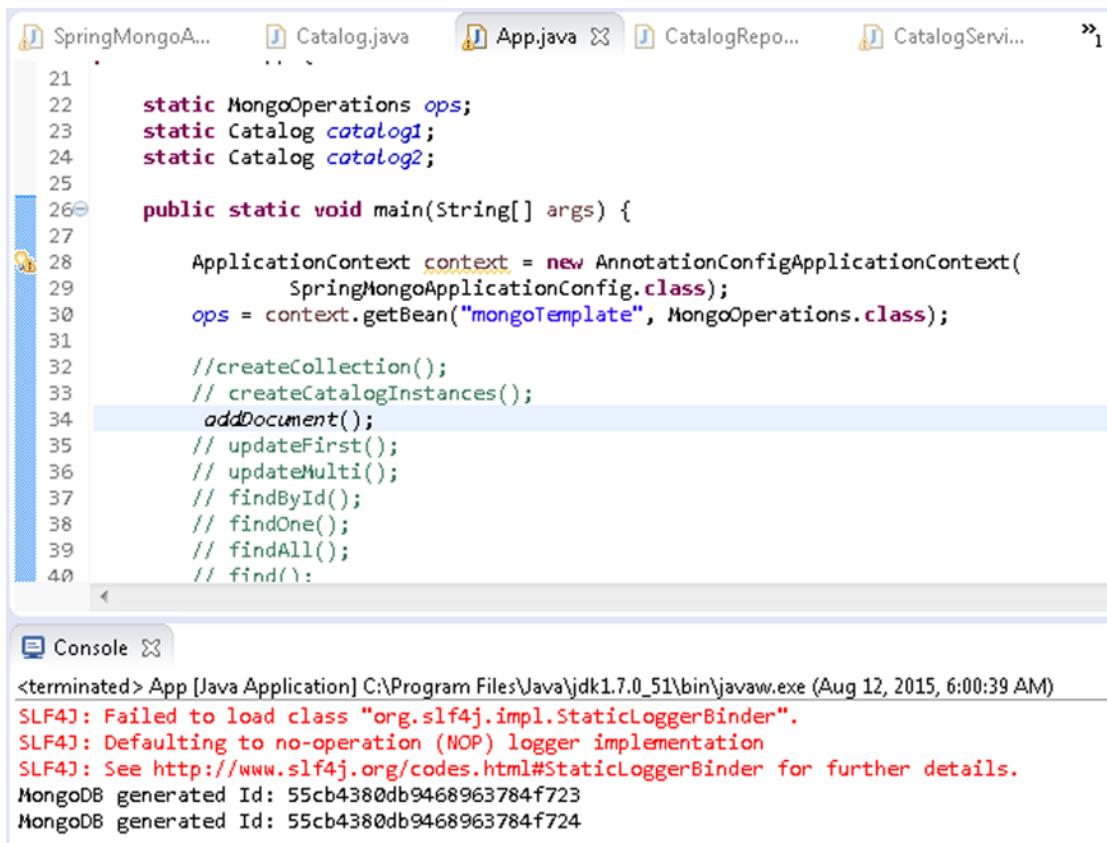
The `addDocument()` method is as follows.

```
private static void addDocument() {
    catalog1 = new Catalog("catalog1", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013",
        "Engineering as a Service", "David A. Kelly");
    ops.save(catalog1, "catalog");
    //ops.save(catalog1);
    System.out.println("MongoDB generated Id: " + catalog1.getId());

    catalog2 = new Catalog("catalog2", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013",
        "Quintessential and Collaborative", "Tom Haunert");
    ops.save(catalog2, "catalog");
    System.out.println("MongoDB generated Id: " + catalog2.getId());

}
```

When the `App.java` application is run with method invocation of the `addDocument()` method, two documents get added to the `catalog` collection. The automatically generated `_id` field values get output the Eclipse Console as shown in Figure 10-8.



The screenshot shows an IDE interface with several tabs at the top: SpringMongoA..., Catalog.java, App.java (which is currently selected), CatalogRepo..., CatalogServ..., and a file ending in .java. Below the tabs is a code editor containing Java code. The code imports org.springframework.context.annotation.AnnotationConfigApplicationContext and org.springframework.data.mongodb.core.MongoOperations. It defines static fields ops, catalog1, and catalog2. The main() method creates a context, gets the mongoTemplate bean, and then calls addDocument(), updateFirst(), updateMulti(), findId(), findOne(), findAll(), and find(). A tooltip for 'addDocument()' is visible. At the bottom, there is a 'Console' tab showing the output of the application's execution.

```

21
22     static MongoOperations ops;
23     static Catalog catalog1;
24     static Catalog catalog2;
25
26     public static void main(String[] args) {
27
28         ApplicationContext context = new AnnotationConfigApplicationContext(
29             SpringMongoApplicationConfig.class);
30         ops = context.getBean("mongoTemplate", MongoOperations.class);
31
32         //createCollection();
33         // createCatalogInstances();
34         addDocument();
35         // updateFirst();
36         // updateMulti();
37         // findById();
38         // findOne();
39         // findAll();
40         // find();
    
```

Console

```

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:00:39 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
MongoDB generated Id: 55cb4380db9468963784f723
MongoDB generated Id: 55cb4380db9468963784f724
    
```

Figure 10-8. Adding Documents

- To list the two documents added, run the db.catalog.find() method in Mongo shell as shown in Figure 10-9.



The screenshot shows the Mongo Shell command line. The user has run the 'show collections' command, which lists 'catalog', 'startup_log', and 'system.indexes'. Then, the user runs 'db.catalog.find()' and gets the following results:

```

> show collections
catalog
startup_log
system.indexes
> db.catalog.find()
[{"_id": ObjectId("55cb4380db9468963784f723"), "_class": "com.mongo.model.Catalog", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November-December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("55cb4380db9468963784f724"), "_class": "com.mongo.model.Catalog", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November-December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}]
> 
    
```

Figure 10-9. Listing Documents in Mongo Shell

The MongoOperations interface also provides the overloaded insert() method to add a single document as discussed in Table 10-6.

Table 10-6. Overloaded insert() Methods

Method	Description
insert(Object objectToSave)	Adds an object (document) to a collection for the entity type of the object to save.
insert(Object objectToSave, String collectionName)	Adds an object (document) to a specified collection.

Adding a Document Batch

In this section we shall add a batch of documents instead of a single document. The MongoOperations interface provides the overloaded `insert()` methods and `insertAll()` method to add or insert a batch of objects to a collection as discussed in Table 10-7. We shall demonstrate each of the three methods in Table 10-7 to add a list of documents to a collection.

Table 10-7. Overloaded Batch Insert Methods Methods

Method	Description
insert(Collection<? extends Object> batchToSave, Class<?> entityClass)	Adds a list of objects of the specified entity class type in a single batch.
insert(Collection<? extends Object> batchToSave, String collectionName)	Adds a list of objects to the specified collection in a single batch.
insertAll(Collection<? extends Object> objectsToSave)	Adds a mixed collection of objects (documents) to a database collection.

1. Add a batch of documents using the `addDocumentBatch()` custom method in the App application.
2. Create an `ArrayList` of `Catalog` instances. Create new `Catalog` instances `catalog1` and `catalog2`. The `Catalog` instances `catalog1` and `catalog2` could be the same as those created in the `addDocument()` custom method. As `catalog1` and `catalog2` are class variables, the same instances may be reused.

```

catalog1 = new Catalog("catalog1", "Oracle Magazine",
"Oracle Publishing", "November-December 2013",
"Engineering as a Service", "David A. Kelly");
catalog2 = new Catalog("catalog2", "Oracle Magazine",
"Oracle Publishing", "November-December 2013",
"Quintessential and Collaborative", "Tom Haunert");
ArrayList arrayList = new ArrayList();
arrayList.add(catalog1);
arrayList.add(catalog2);
    
```

3. Next, use one of the following methods to add a batch of documents:
- Add the `ArrayList` instance using the `insert(Collection<? extends Object> batchToSave, String collectionName)` method to the `catalog` collection.
- ```
ops.insert(arrayList, "catalog");
```
- Alternatively add the batch of objects using the `insert(Collection<? extends Object> batchToSave, Class<?> entityClass)` method.
- ```
ops.insert(arrayList,Catalog.class);
```
- Or the `insertAll(Collection<? extends Object> objectsToSave)` method may be used to add the `ArrayList`. The database collection name to use is determined based on the class.
- ```
ops.insertAll(arrayList);
```

For all of the `insert()` methods and the `insertAll()` method the collection is created implicitly if not already created.

**Note** The source code includes implementations for each of the two overloaded `insert()` methods and for the `insertAll()` method to add a batch of documents. Only one of the method implementations should be invoked at a time to add a batch of two documents. The other method invocations may be commented out.

4. Remove the `catalog` collection before adding a batch of documents with the following commands in Mongo shell.

```
>use local
>db.catalog.drop()
```

When the App application is run to invoke the `addDocumentBatch()` method a batch of documents gets added to the `catalog` collection.

5. Subsequently run the `db.catalog.find()` method in Mongo shell to list the documents added as shown in Figure 10-10. The `_id` is generated automatically.

```
> db.catalog.drop()
true
> db.catalog.find()
[{"_id": ObjectId("55cb449ddb94fc13d2bf45c2"), "class": "com.mongo.model.Catalog", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November-December 2013", "title": "Engineering as a Service", "author": "David A. Kelly"}, {"_id": ObjectId("55cb449ddb94fc13d2bf45c3"), "class": "com.mongo.model.Catalog", "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "November-December 2013", "title": "Quintessential and Collaborative", "author": "Tom Haunert"}]
```

**Figure 10-10.** Listing Documents Added in Batch

## Finding a Document by Id

In this section we shall find a document by Id. The `MongoCollection` interface provides the overloaded `findById()` methods discussed in Table 10-8 for finding a document by Id.

**Table 10-8.** Overloaded `findById()` Methods

| Method                                                                              | Description                                                                                      |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| <code>findById(Object id, Class&lt;T&gt; entityClass)</code>                        | Returns a document by the given id mapped onto the given entity class.                           |
| <code>findById(Object id, Class&lt;T&gt; entityClass, String collectionName)</code> | Returns a document by the given id from the given collection mapped onto the given entity class. |

In this section we shall find a document by Id using the `findById(Object id, Class<T> entityClass, String collectionName)` method.

1. In the `findById()` method in the App application create a `Catalog` instance `catalog1`.

```
catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");
```

2. Save the `catalog1` instance to the `catalog` collection using the `save()` method.

```
ops.save(catalog1, "catalog");
```

3. Find the document using the `findById()` method using the `catalog1.getId()` method invocation for the `id` argument, `Catalog.class` for the entity class argument, and `catalog` as the collection name.

```
Catalog catalog = ops.findById(catalog1.getId(), Catalog.class, "catalog");
```

Alternatively, use the other `findById()` method.

```
Catalog catalog = ops.findById(catalog1.getId(), Catalog.class);
```

4. Subsequently, output the field values from the `Catalog` instance found by id.

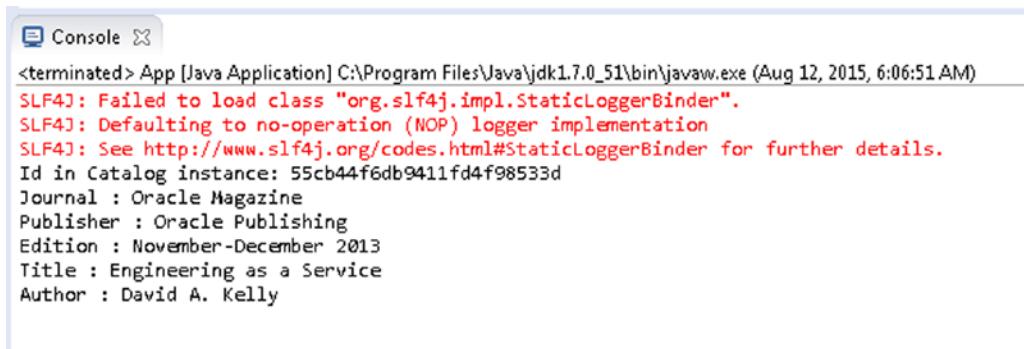
```
System.out.println("Id in Catalog instance: " + catalog1.getId());
System.out.println("Journal : " + catalog.getJournal());
System.out.println("Publisher : " + catalog.getPublisher());
System.out.println("Edition : " + catalog.getEdition());
System.out.println("Title : " + catalog.getTitle());
System.out.println("Author : " + catalog.getAuthor());
```

The `findById()` method in the `App` application is as follows.

```
private static void findById() {
 catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");

 ops.save(catalog1);
 // Catalog catalog = ops.findById(catalog1.getId(),
 // Catalog.class,"catalog");
 Catalog catalog = ops.findById(catalog1.getId(), Catalog.class);
 System.out.println("Id in Catalog instance: " + catalog1.getId());
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
}
```

5. Run the `App.java`. The output lists the field values from the document found by Id as shown in Figure 10-11. What is to be noted is that the id for the `Catalog` instance `catalog1` is not the id value (`catalog1`) specified in the constructor, but the automatically generated id value.



```
Console ×
<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:06:51 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Id in Catalog instance: 55cb44f6db9411fd4f98533d
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly
```

**Figure 10-11.** Finding Documents By Id

## Finding One Document

Another method used to find a single document is the overloaded `findOne()`, which has the following variants, discussed in Table 10-9.

**Table 10-9.** Overloaded `findOne()` Methods

| Method                                                                               | Description                                                                                                                                      |
|--------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>findOne(Query query, Class&lt;T&gt; entityClass)</code>                        | Finds a single instance of an object of the specified entity class type from the collection for the entity class type using the specified query. |
| <code>findOne(Query query, Class&lt;T&gt; entityClass, String collectionName)</code> | Finds a single instance of an object of the specified type from the specified collection using the specified query.                              |

In this section we shall find a single document of type `Catalog` using a `Query` object from the `catalog` collection in the `findOne()` method in the `App` application. First, we need to construct the `Query` object to find the single document. The `BasicQuery` class extends `Query` and provides the following constructors, discussed in Table 10-10.

**Table 10-10.** Overloaded `BasicQuery Constructors`

| Method                                                                                       | Description                                                                                                                                    |
|----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>BasicQuery(com.mongodb.DBObject queryObject)</code>                                    | Creates a <code>BasicQuery</code> instance using the specified <code>DBObject</code> query object.                                             |
| <code>BasicQuery(com.mongodb.DBObject queryObject, com.mongodb.DBObject fieldsObject)</code> | Creates a <code>BasicQuery</code> instance using the specified <code>DBObject</code> query object and the <code>DBObject</code> fields object. |
| <code>BasicQuery(String query)</code>                                                        | Creates a <code>BasicQuery</code> instance using the specified query String.                                                                   |
| <code>BasicQuery(String query, String fields)</code>                                         | Creates a <code>BasicQuery</code> instance using the specified query String and fields String.                                                 |

1. Drop any previously created collection called `catalog` using the following JavaScript method in Mongo shell.

```
>db.catalog.drop()
```

2. In the `findOne()` method in `App` application invoke the `createCatalogInstances()` method to create `Catalog` instances and save the entity instances using the `save()` method.

```
createCatalogInstances();
ops.save(catalog1);
ops.save(catalog2);
```

3. Create a `BasicDBObject` instance using the `BasicDBObject(String key, Object value)` constructor. Specify key as `id` and value as the `ObjectId` instance created from the `id` in the `catalog1` instance obtained using the `getId()` method.

```
DBObject dbObject = new BasicDBObject("id", new ObjectId(catalog1.getId()));
```

4. Using the BasicDBObject instance create a BasicQuery object.

```
BasicQuery query = new BasicQuery(dbObject);
```

5. Using the BasicQuery object find a single document from the catalog collection of entity class type Catalog using either of the findOne() methods.

```
Catalog catalog = ops.findOne(query, Catalog.class);
// Catalog catalog = ops.findOne(query, Catalog.class, "catalog");
```

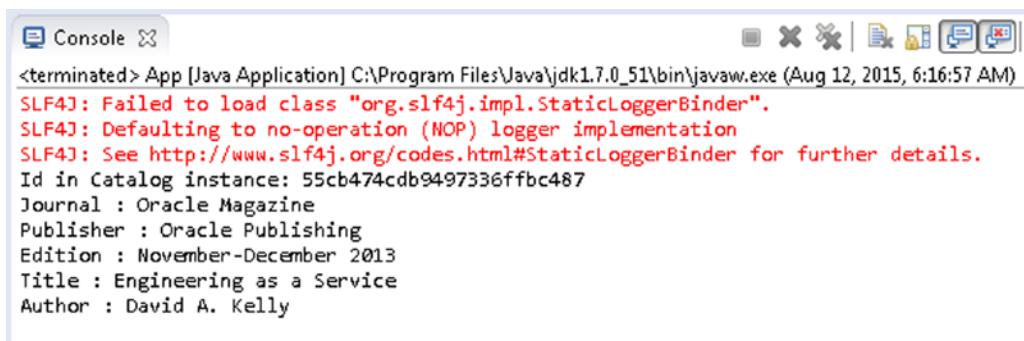
The findOne() method in App application is as follows.

```
private static void findOne() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 DBObject dbObject = new BasicDBObject("id", new ObjectId(
 catalog1.getId()));
 BasicQuery query = new BasicQuery(dbObject);

 Catalog catalog = ops.findOne(query, Catalog.class);
 // Catalog catalog = ops.findOne(query, Catalog.class, "catalog");
 System.out.println("Id in Catalog instance: " + catalog.getId());
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());

}
```

6. Run the App application to save some Catalog instances and find one of the Catalog instances using the findOne() method in MongoOperations. The output from the findOne() method is shown in the Eclipse Console in Figure 10-12.



The screenshot shows the Eclipse IDE's Console view. The title bar says "Console". The console output is as follows:

```
<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:16:57 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Id in Catalog instance: 55cb474cdb9497336ffbc487
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly
```

**Figure 10-12.** Finding a Document with findOne()

A new Query instance to be given to the `findOne()` method may also be created using a criteria definition with the `Query(CriteriaDefinition criteriaDefinition)` constructor. The `Criteria` class implements the `CriteriaDefinition` interface and provides several methods to create a criterion and return a `Criteria` instance. Use the `where(String key)` method to specify a key for a criterion and subsequently invoke the `is(Object o)` method to compare the key to the `id` field value in the `Catalog2 Catalog` instance.

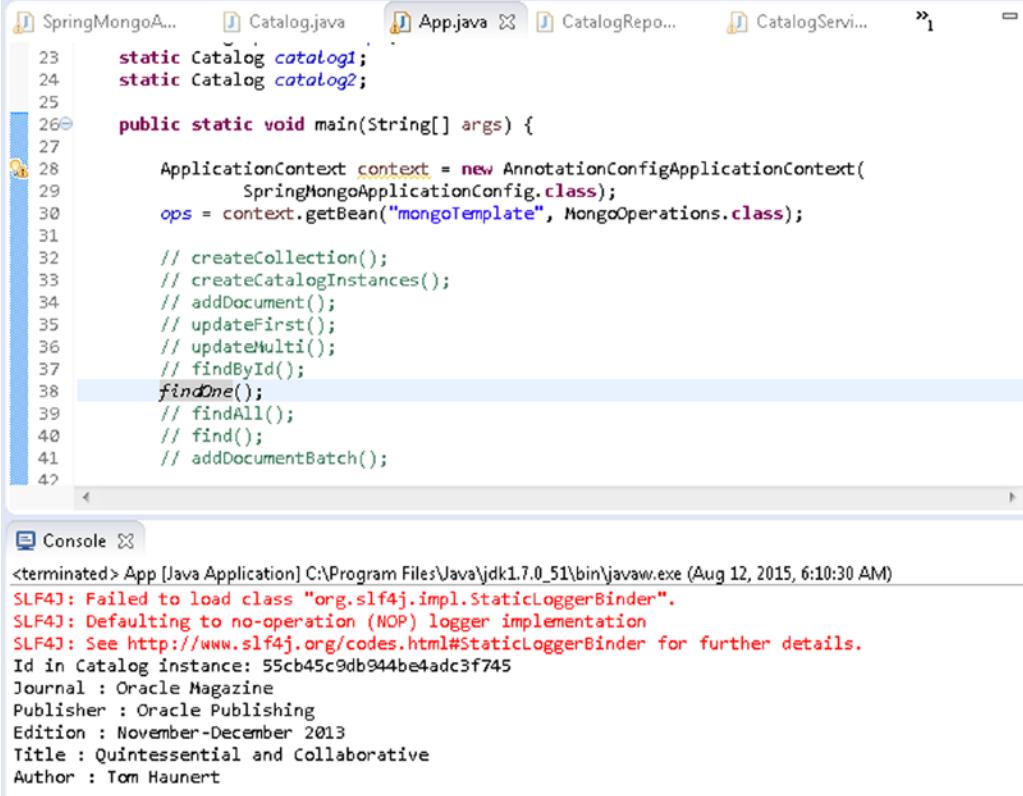
7. Using the `Criteria` instance returned by the `sequence` method invocation of `where()` and `is()` methods creates a `Query` instance and using the `Query` instance invokes the `findOne(Query query, Class<T> entityClass)` method.

```
String _id = catalog2.getId();
Catalog catalog = ops.findOne(new Query(Criteria.where("_id").is(_id)),
Catalog.class);
```

8. Output the field values in the `Catalog` instance returned by the `findOne()` method.

```
System.out.println("Id in Catalog instance: " + catalog2.getId());
System.out.println("Journal : " + catalog.getJournal());
System.out.println("Publisher : " + catalog.getPublisher());
System.out.println("Edition : " + catalog.getEdition());
System.out.println("Title : " + catalog.getTitle());
System.out.println("Author : " + catalog.getAuthor());
```

When the App application is run the field values in the `Catalog` instance found by using a criteria definition are output as shown in Figure 10-13.



The screenshot shows an IDE interface with several tabs at the top: SpringMongoA..., Catalog.java, App.java (which is the active tab), CatalogRepo..., CatalogServic..., and a file ending in ...1. Below the tabs is a code editor containing Java code. The code defines two static Catalog objects, catalog1 and catalog2, and a main method. The main method creates an ApplicationContext, gets a MongoTemplate bean, and then calls the findOne() method on it. The findOne() call is highlighted with a blue selection bar. The code also includes comments for other methods like createCollection(), createCatalogInstances(), addDocument(), updateFirst(), updateMulti(), findbyId(), findAll(), find(), and addDocumentBatch(). In the bottom right corner of the code editor, there is a small icon of a person with a gear.

```

23 static Catalog catalog1;
24 static Catalog catalog2;
25
26 public static void main(String[] args) {
27
28 ApplicationContext context = new AnnotationConfigApplicationContext(
29 SpringMongoApplicationConfig.class);
30 ops = context.getBean("mongoTemplate", MongoOperations.class);
31
32 // createCollection();
33 // createCatalogInstances();
34 // addDocument();
35 // updateFirst();
36 // updateMulti();
37 // findbyId();
38 findOne();
39 // findAll();
40 // find();
41 // addDocumentBatch();
42

```

**Console**

```

terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:10:30 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Id in Catalog instance: 55cb45c9db944be4adc3f745
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-13.** Finding a Document with Query Criteria

## Finding All Documents

The MongoOperations interface provides the overloaded `findAll()` method to find all documents from a collection as discussed in Table 10-11.

**Table 10-11.** Overloaded `findAll()` Methods

| Method                                                                  | Description                                                                                   |
|-------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <code>findAll(Class&lt;T&gt; entityClass)</code>                        | Returns a list of documents from the collection for the specified entity class type.          |
| <code>findAll(Class&lt;T&gt; entityClass, String collectionName)</code> | Returns a list of documents from the specified collection of the specified entity class type. |

1. In the `findAll()` method in App application first add some Catalog instances to the catalog collection.
2. Subsequently find all documents of entity class type Catalog using the `findAll(Class<T> entityClass)` method. The `findAll()` method returns a `List` instance.
3. Obtain a `Iterator<Catalog>` from the `List` instance using the `iterator()` method.

```
Iterator<Catalog> iter = list.iterator();
```

4. Iterate over the result set using the `hasNext()` method in a `while` loop and obtain the Catalog instances in the result set.
5. Output the field values in the Catalog instances using the `get()` methods for the fields defined in the Catalog class. The `findAll()` method in App class is as follows.

```
private static void findAll() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}
```

When the App application is run the field values from all the Catalog instances get output to the Console as shown in Figure 10-14.

The screenshot shows an IDE interface with several tabs at the top: 'SpringMongoA...', 'Catalog.java', '\*App.java', 'CatalogRepo...', and '»2'. The code editor displays Java code for a class named 'App' which uses Spring's MongoDB annotations. The code includes imports for 'MongoOperations', 'Catalog', and 'MongoTemplate'. It demonstrates various MongoDB operations like creating collections, adding documents, and finding all documents. Below the code editor is a 'Console' tab showing the application's output. The output starts with a log message from SLF4J indicating it is defaulting to a NOP logger implementation. It then lists several document details, such as 'Journal : Oracle Magazine', 'Publisher : Oracle Publishing', and 'Title : Engineering as a Service'. This output corresponds to the data shown in the 'Catalog' class.

```

public class App {
 static MongoOperations ops;
 static Catalog catalog1;
 static Catalog catalog2;

 public static void main(String[] args) {
 ApplicationContext context = new AnnotationConfigApplicationContext(
 SpringMongoApplicationConfig.class);
 ops = context.getBean("mongoTemplate", MongoOperations.class);

 // createCollection();
 // createCatalogInstances();
 // addDocument();
 // updateFirst();
 // updateMulti();
 // findById();
 // findOne();
 findAll();
 }
}

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:18:33 AM)
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-14.** Finding All Documents

## Finding Documents Using a Query

In the preceding sections we have found documents using `findAll` (to find all documents), `findOne` (to find a single document), and `findById` (to find by Id). The `MongoOperations` interface provides the overloaded `find()` methods to find document/s using a specific query. The `find()` methods return a `List<T>` instance and are discussed in Table 10-12.

**Table 10-12.** Overloaded find() Methods

| Method                                                         | Description                                                                                                          |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| find(Query query, Class<T> entityClass)                        | Return a list of documents from the collection for the specified entity class type for the specified query.          |
| find(Query query, Class<T> entityClass, String collectionName) | Return a list of documents from the specified collection of the specified entity class type for the specified query. |

In the `find()` method in the App application we shall find documents in which the `publisher` field value is Oracle Publishing.

1. Create a `BasicDBObject` instance using the constructor `BasicDBObject(String key, Object value)` with key as `publisher` and value as Oracle Publishing.

```
DBObject dbObject = new BasicDBObject("publisher," "Oracle Publishing");
```

2. Create a `BasicQuery` object from the `BasicDBObject` object using constructor `BasicQuery(com.mongodb.DBObject queryObject)`.

```
BasicQuery query = new BasicQuery(dbObject);
```

3. Using the `BasicQuery` instance invoke the `find(Query query, Class<T> entityClass, String collectionName)` method to find documents for the specified query. The `find()` method returns a `List<Catalog>` instance of documents.

```
List<Catalog> list = ops.find(query, Catalog.class, "catalog");
```

4. Obtain an iterator from the `List` instance and iterate over the `List` instance to output the field values from the documents in the list. The `find()` method in App application is as follows.

```
private static void find() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 DBObject dbObject = new BasicDBObject("publisher", "Oracle Publishing");
 BasicQuery query = new BasicQuery(dbObject);
 List<Catalog> list = ops.find(query, Catalog.class, "catalog");
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}
```

5. Run the App application to output the field values from the documents found using the `find(Query query, Class<T> entityClass, String collectionName)` method as shown in Figure 10-15.

```

Catalog.java
172 private static void find() {
173 createCatalogInstances();
174 ops.save(catalog1);
175 ops.save(catalog2);
176
177 DBObject dbObject = new BasicDBObject("publisher", "Oracle Publishing");
178 BasicQuery query = new BasicQuery(dbObject);
179 List<Catalog> list = ops.find(query, Catalog.class, "catalog");
180
181 /*List<Catalog> list = ops.find(
182 new Query(Criteria.where("journal").is("Oracle Magazine")),
183 Catalog.class);*/
184 Iterator<Catalog> iter = list.iterator();
185 while (iter.hasNext()) {
186 Catalog catalog = iter.next();
187 System.out.println("Journal : " + catalog.getJournal());
188 System.out.println("Publisher : " + catalog.getPublisher());
189 System.out.println("Edition : " + catalog.getEdition());
190 System.out.println("Title : " + catalog.getTitle());
191 }
192 }

```

```

Console
<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 11:43:31 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-15.** Finding Documents with `find()`

6. As was discussed for the `findOne(Query query, Class<T> entityClass)` method a criteria definition may also be used to find documents. Create a Criteria using the where and is method invocations in sequence to create a Criteria instance for journal field value as Oracle Magazine. Create a Query instance from the Criteria instance and invoke the `find()` method using the Query instance and the Catalog.class entity class type.

```
List<Catalog> list = ops.find(new Query(Criteria.where("journal").is("Oracle Magazine")), Catalog.class);
```

7. Iterate over the list returned by the `find` method to output the field values for the `Catalog` instances in the list. The output from the `find()` method using a criterion is shown in Eclipse Console in Figure 10-16.

The screenshot shows the Eclipse IDE interface. In the top editor area, the `Catalog.java` file is open, displaying Java code that creates two catalog instances and then finds catalog documents where the journal is "Oracle Magazine". The code uses the `ops.find` method with a query specifying the `journal` field. In the bottom console window, the application's output is shown, including SLF4J initialization messages and the printed details of the found catalog documents.

```

172 private static void find() {
173 createCatalogInstances();
174 ops.save(catalog1);
175 ops.save(catalog2);
176
177 /*DBObject dbObject = new BasicDBObject("publisher", "Oracle Publishing");
178 BasicQuery query = new BasicQuery(dbObject);
179 List<Catalog> list = ops.find(query, Catalog.class, "catalog");*/
180
181 List<Catalog> list = ops.find(
182 new Query(Criteria.where("journal").is("Oracle Magazine")),
183 Catalog.class);
184 Iterator<Catalog> iter = list.iterator();
185 while (iter.hasNext()) {
186 Catalog catalog = iter.next();
187 System.out.println("Journal : " + catalog.getJournal());
188 System.out.println("Publisher : " + catalog.getPublisher());
189 System.out.println("Edition : " + catalog.getEdition());
190 System.out.println("Title : " + catalog.getTitle());
191 System.out.println("Author : " + catalog.getAuthor());
192 }

```

Console Output:

```

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 11:46:28 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-16.** Finding Documents with Query Criteria with `find()` Method

## Updating the First Document

In this section we shall update a document using a query to find the document to update. A query could return multiple documents. If only the first document in the query result is to be updated the `MongoOperations` interface provides the overloaded `updateFirst()` method. The `org.springframework.data.mongodb.core.query.Update` class is used to provide the update clauses. Each of the `updateFirst()` methods, detailed in Table 10-13, returns a `WriteResult` object.

**Table 10-13.** Overloaded updateFirst() Methods

| Method                                                                               | Description                                                                                                                                     |
|--------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| updateFirst(Query query, Update update, Class<?> entityClass)                        | Updates the first document of the given entity class type that is found using the given query with the update document.                         |
| updateFirst(Query query, Update update, Class<?> entityClass, String collectionName) | Updates the first document of the given entity class type that is found in the given collection using the given query with the update document. |
| updateFirst(Query query, Update update, String collectionName)                       | Updates the first document that is found in the given collection using the given query with the update document.                                |

1. Remove any previously added documents using the `db.catalog.drop()` method in Mongo shell.
2. In the `updateFirst()` method in App class save some Catalog instances using the `createCatalogInstances()` method to create the Catalog instances and the `save()` method to save the Catalog instances.
3. Create a `BasicDBObject` instance for `edition` field as key and November-December 2013 as value.

```
DBObject dbObject = new BasicDBObject("edition", "November-December 2013");
```

4. Create a `BasicQuery` instance from the `BasicDBObject` instance.

```
BasicQuery query = new BasicQuery(dbObject);
```

5. Invoke the `updateFirst(Query query, Update update, Class<?> entityClass)` method using the `BasicQuery` object as the first argument. Create the `Update` object for the second argument using the `Update` class static method `update(String key, Object value)` with key as `edition` and value as `11-12-2013`, which implies that the `edition` field is to be updated to `11-12-2013`. For the third argument specify `Catalog.class`. Output the `WriteResult` object returned by the `updateFirst()` method.

```
WriteResult result = ops.updateFirst(query, Update.update("edition", "11-12-2013"), Catalog.class); System.out.println(result);
```

6. Subsequent to updating invoke the `findAll(Class<T> entityClass)` method to find all the documents, iterate over the list of documents found and output their field values.

```
List<Catalog> list = ops.findAll(Catalog.class);
Iterator<Catalog> iter = list.iterator();
while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
```

```

 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}

```

The updateFirst() method in the App class is as follows.

```

private static void updateFirst() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 DBObject dbObject = new BasicDBObject("edition",
 "November-December 2013");
 BasicQuery query = new BasicQuery(dbObject);
 WriteResult result = ops.updateFirst(query,
 Update.update("edition", "11-12-2013"), Catalog.class);
 System.out.println(result);
 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}

```

When the App application is run to invoke the updateFirst() method the first document gets updated and subsequent listings of the documents found using the find() method include the edition field in the first document updated to 11-12-2013 as was specified in the update document as shown in Figure 10-17. The other documents still have the edition field as November-December 2013. The WriteResult field n has value 1, which implies that one document has been updated.

The screenshot shows an IDE interface with several tabs at the top: SpringMongoA..., Catalog.java, App.java (which is currently selected), CatalogRepo..., and two others. The code in App.java is as follows:

```

20 public class App {
21
22 static MongoOperations ops;
23 static Catalog catalog1;
24 static Catalog catalog2;
25
26 public static void main(String[] args) {
27
28 ApplicationContext context = new AnnotationConfigApplicationContext(
29 SpringMongoApplicationConfig.class);
30 ops = context.getBean("mongoTemplate", MongoOperations.class);
31
32 // createCollection();
33 // createCatalogInstances();
34 // addDocument();
35 updateFirst();
36 // updateMulti();
37 // findById();
38 // findOne();

```

Below the code editor is a 'Console' tab with the following output:

```

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:24:38 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 1 , "updatedExisting" : true}
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : 11-12-2013
Title : Engineering as a Service
Author : David A. Kelly
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-17.** Updating a Document

The query may also be specified using a criteria definition as follows.

```
WriteResult result = ops.updateFirst(new Query(Criteria.where("edition").is("November-December 2013")), Update.update("edition", "11-12-2013"), Catalog.class);
```

## Update Multiple Documents

If multiple documents are to be updated the MongoOperations interface provides the overloaded updateMulti() method as discussed in Table 10-14.

**Table 10-14.** Overloaded updateMulti() Methods

| Method                                                                               | Description                                                                                                                                 |
|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| updateMulti(Query query, Update update, Class<?> entityClass)                        | Updates all documents of the given entity class type that are found using the given query with the given update document.                   |
| updateMulti(Query query, Update update, Class<?> entityClass, String collectionName) | Updates all documents of the given entity class type that are found in the given collection using the given query with the update document. |
| updateMulti(Query query, Update update, String collectionName)                       | Updates all documents that are found in the given collection using the given query with the update document.                                |

Next, we shall update multiple documents in the catalog collection in the App class method updateMulti().

1. Drop the catalog collection using the db.catalog.drop() method in Mongo shell.
2. Create new documents using the createCatalogInstances() method and save the documents using the save() method.
3. Create a BasicQuery object for the query document. The query document selects all documents with edition field as November-December 2013.

```
DBObject dbObject = new BasicDBObject("edition", "November-December 2013");
BasicQuery query = new BasicQuery(dbObject);
```

4. Invoke the updateMulti(Query query, Update update, Class<?> entityClass) method with the BasicQuery object as the first argument. Create a Update instance for edition field with value 11-12-2013 using the static method update(String key, Object value) for the second argument. Use entity class Catalog.class for the third argument. Output the WriteResult object returned by the updateMulti() method.

```
WriteResult result = ops.updateMulti(query, Update.update("edition",
"11-12-2013"), Catalog.class);
System.out.println(result);
```

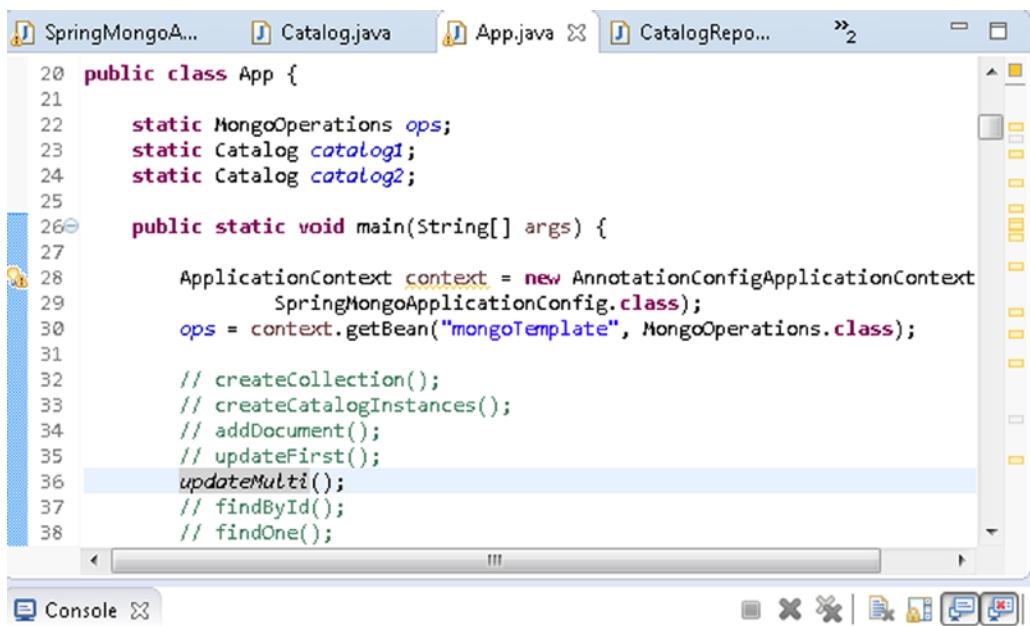
5. Subsequently invoke the findAll() method to find and output field values from all the documents to verify that the documents got updated. The updateMulti() method is as follows.

```
private static void updateMulti() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 DBObject dbObject = new BasicDBObject("edition",
 "November-December 2013");
 BasicQuery query = new BasicQuery(dbObject);
 WriteResult result = ops.updateMulti(query,
```

```
 Update.update("edition", "11-12-2013"), Catalog.class);
System.out.println(result);
List<Catalog> list = ops.findAll(Catalog.class);
Iterator<Catalog> iter = list.iterator();
while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
}
}
```

6. Drop any previously added catalog collection with the db.catalog.drop() method in Mongo shell.

When the App application is run to invoke the updateMulti() method, all documents for the specified query get updated. Subsequent invocation of findAll() lists the modified documents in which the edition field has been updated in all the documents, not just the first document as shown in Figure 10-18.



```

20 public class App {
21
22 static MongoOperations ops;
23 static Catalog catalog1;
24 static Catalog catalog2;
25
26 public static void main(String[] args) {
27
28 ApplicationContext context = new AnnotationConfigApplicationContext(
29 SpringMongoApplicationConfig.class);
30 ops = context.getBean("mongoTemplate", MongoOperations.class);
31
32 // createCollection();
33 // createCatalogInstances();
34 // addDocument();
35 // updateFirst();
36 updateMulti();
37 // findById();
38 // findOne();

```

**Console**

```

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:25:54 AM)
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 2 , "updatedExisting" : true}
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : 11-12-2013
Title : Engineering as a Service
Author : David A. Kelly
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : 11-12-2013
Title : Quintessential and Collaborative
Author : Tom Haunert

```

**Figure 10-18.** Updating Documents with `updateMulti()` Method

The query document in the `updateMulti()` method invocation may also be created using a criteria definition as follows.

```
WriteResult result = ops.updateMulti(new Query(Criteria.where("edition").is("November-December 2013")), Update.update("edition", "11-12-2013"), Catalog.class);
```

## Removing Documents

In this section we shall remove documents from the `catalog` collection. The `MongoOperations` interface provides the overloaded `remove()` methods to remove documents for a specified query as discussed in Table 10-15.

**Table 10-15.** Overloaded remove() Methods

| Method                                                           | Description                                                                                                |
|------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| remove(Query query, Class<?> entityClass)                        | Removes all documents that match the specified query from the collection for the given entity class.       |
| remove(Query query, Class<?> entityClass, String collectionName) | Removes all documents that match the specified query from the given collection for the given entity class. |
| remove(Query query, String collectionName)                       | Removes all documents that match the specified query from the given collection.                            |

The MongoOperations interface also provides the overloaded `findAndRemove()` method to find and remove a single document as discussed in Table 10-16.

**Table 10-16.** Overloaded findAndRemove() Methods

| Method                                                                  | Description                                                                                                                                                   |
|-------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| findAndRemove(Query query, Class<T> entityClass)                        | Finds and removes a single document from the collection for the given entity class. The first document that matches the given query is removed.               |
| findAndRemove(Query query, Class<T> entityClass, String collectionName) | Finds and removes a single document of the specified entity class type from the given collection. The first document that matches the given query is removed. |

First, we shall look at an example using the `remove()` method.

1. In the custom `remove()` class method in the App application (not to be confused with the `remove()` method in the MongoOperations interface), create and save two Catalog instances.
2. Create a `BasicDBObject` instance using the `_id` field as key and `id` field value in the `catalog1` instance as value. Create a `BasicQuery` instance using the `BasicDBObject` instance.

```
DBObject dbObject = new BasicDBObject("_id", catalog1.getId());
BasicQuery query = new BasicQuery(dbObject);
```

3. Invoke one of the `remove()` methods to remove all the documents that match the specified query. Output the result of the `remove()` method.

```
WriteResult result = ops.remove(query, Catalog.class);
//WriteResult result = ops.remove(query, "catalog");
//WriteResult result =ops.remove(query, Catalog.class, "catalog");
System.out.println(result);
```

4. Subsequently invoke the `findAll()` method to find and list all the documents. When the App application is run to invoke the `remove()` method the documents that match the given query get removed. For the example, one of the two documents gets removed and the other gets listed with `findAll()` as shown in Figure 10-19.

The screenshot shows an IDE interface with several tabs at the top: SpringMongoA..., Catalog.java, App.java, CatalogRepo... (which is the active tab), and a few others. The CatalogRepo... tab contains Java code for a MongoDB repository. The code includes imports for ApplicationContext, AnnotationConfigApplicationContext, SpringMongoApplicationConfig, MongoOperations, and MongoTemplate. It also includes annotations like @Repository and @Query. Below the code, there's a large block of commented-out methods: createCollection(), createCatalogInstances(), addDocument(), updateFirst(), updateMulti(), findById(), findOne(), findAll(), find(), and addDocumentBatch(). At the bottom of the code area, there's a call to remove().

**Console Output:**

```
<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:37:38 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 1}
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Quintessential and Collaborative
Author : Tom Haunert
```

**Figure 10-19.** Removing a Single Document with `remove()`

- To remove all documents the following query may be used.

```
DBObject dbObject = new BasicDBObject("edition","November-December 2013");
BasicQuery query = new BasicQuery(dbObject);
```

When the `remove()` method is invoked with the preceding query the two documents added get removed as indicated by the `n` field value of 2 as shown in Figure 10-20.

The screenshot shows an IDE interface with the same tabs as Figure 10-19. The CatalogRepo... tab contains the same Java code as Figure 10-19. The console output below shows the removal of two documents. The output starts with the terminated application message, followed by SLF4J logging, and then the results of the remove operation: a document with "serverUsed": "localhost:27017", "ok": 1, and "n": 2.

```
<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:39:58 AM)
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 2}
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

**Figure 10-20.** Removing Multiple Documents with `remove()`

The `remove()` method in `App` application is as follows.

```
private static void remove() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 //DBObject dbObject = new BasicDBObject("edition", "November-December 2013");
 DBObject dbObject = new BasicDBObject("_id", catalog1.getId());
 BasicQuery query = new BasicQuery(dbObject);
 WriteResult result = ops.remove(query, Catalog.class);
 // WriteResult result = ops.remove(query, "catalog");
 System.out.println(result);

 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}
```

Next, we shall demonstrate the `findAndModify()` method.

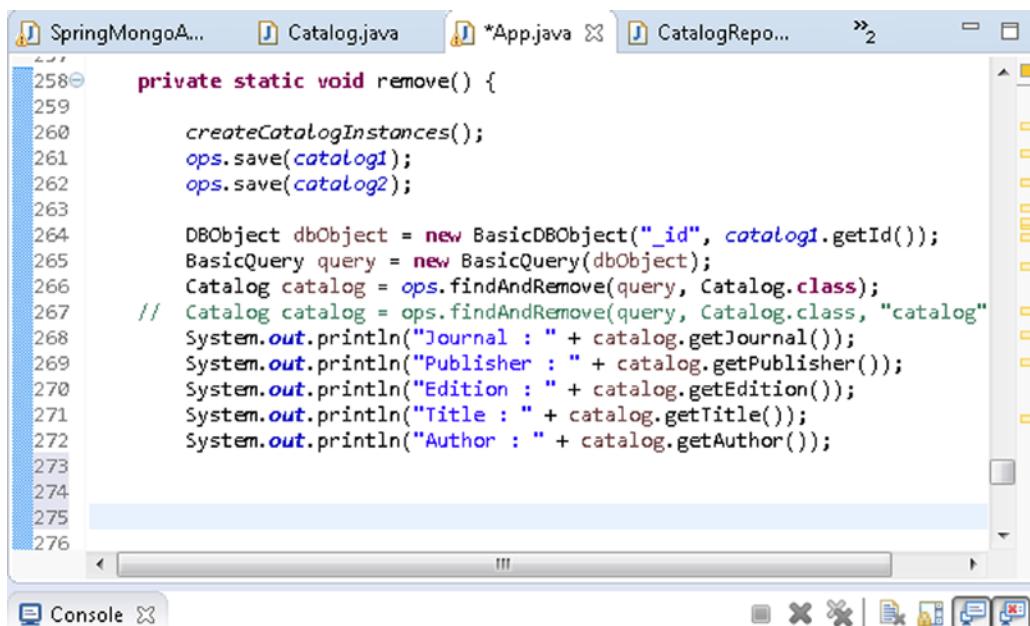
1. Create a `BasicQuery` object to find all documents with a specific the `_id` field value. The query should return only one document as the `_id` field value is unique.

```
DBObject dbObject = new BasicDBObject("_id", catalog1.getId());
BasicQuery query = new BasicQuery(dbObject);
```

2. Invoke one of the `findAndRemove()` methods using the query.

```
Catalog catalog = ops.findAndRemove(query, Catalog.class);
//Catalog catalog = ops.findAndRemove(query, Catalog.class, "catalog");
```

3. The `findAndRemove()` method returns the document that is found and removed. Output the field values from the removed document. When the `App` application is run the document matching the query is removed and its field values are output to the Eclipse Console as shown in Figure 10-21.



The screenshot shows an IDE interface with several tabs at the top: 'SpringMongoA...', 'Catalog.java', '\*App.java', 'CatalogRepo...', and '»2'. The code editor displays Java code for a Catalog class, specifically a method named 'remove()'. The code uses Spring Data annotations like @Repository and @Service. It creates two Catalog instances, saves them, and then attempts to remove them using the 'findAndRemove' method. A comment indicates an alternative approach using 'findAndDelete'. Below the code editor is a 'Console' tab showing the application's output. The output shows SLF4J logging messages indicating it failed to load a specific logger binder and defaulted to a no-operation (NOP) implementation. It then prints the details of a Catalog object: Journal (Oracle Magazine), Publisher (Oracle Publishing), Edition (November-December 2013), Title (Engineering as a Service), and Author (David A. Kelly).

```

private static void remove() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);

 DBObject dbObject = new BasicDBObject("_id", catalog1.getId());
 BasicQuery query = new BasicQuery(dbObject);
 Catalog catalog = ops.findAndRemove(query, Catalog.class);
 // Catalog catalog = ops.findAndRemove(query, Catalog.class, "catalog");
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
}

```

```

<terminated> App [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015, 6:47:24 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal : Oracle Magazine
Publisher : Oracle Publishing
Edition : November-December 2013
Title : Engineering as a Service
Author : David A. Kelly

```

**Figure 10-21.** Removing a Document with `findAndRemove()`

The App application is listed below with some of the method invocations and code sections commented out. Uncomment the code sections to test before running the application.

```

package com.mongo.core;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.bson.types.ObjectId;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.mongo.config.SpringMongoApplicationConfig;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.query.BasicQuery;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.core.query.Update;
import com.mongo.model.Catalog;

```

```
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.WriteResult;

public class App {

 static MongoOperations ops;
 static Catalog catalog1;
 static Catalog catalog2;

 public static void main(String[] args) {

 ApplicationContext context = new AnnotationConfigApplicationContext(
 SpringMongoApplicationConfig.class);
 ops = context.getBean("mongoTemplate", MongoOperations.class);

 // createCollection();
 // createCatalogInstances();
 // addDocument();
 // updateFirst();
 // updateMulti();
 // findById();
 // findOne();
 // findAll();
 find();
 // addDocumentBatch();

 //remove();
 }

 private static void createCollection() {
 if (!ops.collectionExists("catalog")) {
 ops.createCollection("catalog");
 } else {
 ops.dropCollection("catalog");
 ops.createCollection("catalog");
 }
 }

 private static void createCatalogInstances() {
 catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");

 catalog2 = new Catalog("catalog2", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Quintessential and Collaborative", "Tom Haunert");
 }
}
```

```

private static void addDocument() {

 catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");
 // ops.save(catalog1); //collection created implicitly
 ops.save(catalog1); // collection created implicitly by same name as
 // object class
 System.out.println("MongoDB generated Id: " + catalog1.getId());

 catalog2 = new Catalog("catalog2", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Quintessential and Collaborative", "Tom Haunert");
 ops.save(catalog2, "catalog");
 System.out.println("MongoDB generated Id: " + catalog2.getId());

}

private static void addDocumentBatch() {

 catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");
 catalog2 = new Catalog("catalog2", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Quintessential and Collaborative", "Tom Haunert");

 ArrayList arrayList = new ArrayList();
 arrayList.add(catalog1);
 arrayList.add(catalog2);
 ops.insert(arrayList, "catalog");

 // ops.insert(arrayList,Catalog.class);

 // ops.insertAll(arrayList);

}

private static void findById() {

 catalog1 = new Catalog("catalog1", "Oracle Magazine",
 "Oracle Publishing", "November-December 2013",
 "Engineering as a Service", "David A. Kelly");

 ops.save(catalog1);

 // Catalog catalog = ops.findById(catalog1.getId(),
 // Catalog.class,"catalog");

 Catalog catalog = ops.findById(catalog1.getId(), Catalog.class);
 System.out.println("Id in Catalog instance: " + catalog1.getId());
 System.out.println("Journal : " + catalog.getJournal());
}

```

```

 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }

private static void findOne() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);

 /*
 * String _id = catalog2.getId(); Catalog catalog = ops.findOne(new
 * Query(Criteria.where("_id").is(_id)), Catalog.class);
 * System.out.println("Id in Catalog instance: " + catalog2.getId());
 * System.out.println("Journal : " + catalog.getJournal());
 * System.out.println("Publisher : " + catalog.getPublisher());
 * System.out.println("Edition : " + catalog.getEdition());
 * System.out.println("Title : " + catalog.getTitle());
 * System.out.println("Author : " + catalog.getAuthor());
 */
}

DBObject dbObject = new BasicDBObject("id", new ObjectId(
 catalog1.getId()));
BasicQuery query = new BasicQuery(dbObject);

Catalog catalog = ops.findOne(query, Catalog.class);
catalog = ops.findOne(query, Catalog.class, "catalog");
System.out.println("Id in Catalog instance: " + catalog1.getId());
System.out.println("Journal : " + catalog.getJournal());
System.out.println("Publisher : " + catalog.getPublisher());
System.out.println("Edition : " + catalog.getEdition());
System.out.println("Title : " + catalog.getTitle());
System.out.println("Author : " + catalog.getAuthor());

}

private static void findAll() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}

```

```

private static void find() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);

 /*DBObject dbObject = new BasicDBObject("publisher", "Oracle Publishing");
 BasicQuery query = new BasicQuery(dbObject);
 List<Catalog> list = ops.find(query, Catalog.class, "catalog");*/
}

List<Catalog> list = ops.find(
 new Query(Criteria.where("journal").is("Oracle Magazine")),
 Catalog.class);
Iterator<Catalog> iter = list.iterator();
while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
}
}

private static void updateFirst() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);

 WriteResult result = ops.updateFirst(new Query(Criteria
 .where("edition").is("November-December 2013")), Update.update(
 "edition", "11-12-2013"), Catalog.class);
 System.out.println(result);

 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
}

/*
 * DBObject dbObject = new BasicDBObject("edition",
 * "November-December 2013"); BasicQuery query = new
 * BasicQuery(dbObject); WriteResult result = ops.updateFirst(query,
 * Update.update("edition", "11-12-2013"), Catalog.class);
 * System.out.println(result);
 */

```

```

 * List<Catalog> list = ops.findAll(Catalog.class); Iterator<Catalog>
 * iter = list.iterator(); while (iter.hasNext()) { Catalog catalog =
 * iter.next(); System.out.println("Journal : " + catalog.getJournal());
 * System.out.println("Publisher : " + catalog.getPublisher());
 * System.out.println("Edition : " + catalog.getEdition());
 * System.out.println("Title : " + catalog.getTitle());
 * System.out.println("Author : " + catalog.getAuthor()); }
 */

 }

private static void updateMulti() {
 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);
 DBObject dbObject = new BasicDBObject("edition",
 "November-December 2013");
 BasicQuery query = new BasicQuery(dbObject);
 WriteResult result = ops.updateMulti(query,
 Update.update("edition", "11-12-2013"), Catalog.class);
 System.out.println(result);
 List<Catalog> list = ops.findAll(Catalog.class);
 Iterator<Catalog> iter = list.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
 }
 /*
 * WriteResult result = ops.updateMulti(new Query(Criteria
 * .where("edition").is("November-December 2013")), Update.update(
 * "edition", "11-12-2013"), Catalog.class); System.out.println(result);
 * findAll();
 */
}
}

private static void remove() {

 createCatalogInstances();
 ops.save(catalog1);
 ops.save(catalog2);

 DBObject dbObject = new BasicDBObject("_id", catalog1.getId());
 BasicQuery query = new BasicQuery(dbObject);
 Catalog catalog = ops.findAndRemove(query, Catalog.class);
// Catalog catalog = ops.findAndRemove(query, Catalog.class, "catalog");
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
}

```

```

System.out.println("Edition : " + catalog.getEdition());
System.out.println("Title : " + catalog.getTitle());
System.out.println("Author : " + catalog.getAuthor());

/*
 * DBObject dbObject = new
 * BasicDBObject("edition", "November-December 2013"); //DBObject
 * dbObject = new BasicDBObject("_id", catalog1.getId()); BasicQuery
 * query = new BasicQuery(dbObject); // WriteResult result =
 * ops.remove(query, Catalog.class); WriteResult result =
 * ops.remove(query, "catalog"); System.out.println(result);
 */

/*List<Catalog> list = ops.findAll(Catalog.class);
Iterator<Catalog> iter = list.iterator();
while (iter.hasNext()) {
 catalog = iter.next();
 System.out.println("Journal : " + catalog.getJournal());
 System.out.println("Publisher : " + catalog.getPublisher());
 System.out.println("Edition : " + catalog.getEdition());
 System.out.println("Title : " + catalog.getTitle());
 System.out.println("Author : " + catalog.getAuthor());
}*/}

}
}

```

## Using Spring Data with MongoDB

Spring Data repositories are an abstraction that implement a data access layer over the underlying datastore. Spring Data repositories reduce the boilerplate code required to access a datastore. Spring Data repositories may be used with MongoDB data store.

To enable the Spring Data repositories infrastructure for MongoDB annotate the JavaConfig class with `@EnableMongoRepositories`. Annotate the JavaConfig class `SpringMongoApplicationConfig` class with `@EnableMongoRepositories("com.mongo.repositories")`. The `com.mongo.repositories` is the package to search for repositories. Specify the packages to scan for annotated components using the `@ComponentScan` annotation with the `basePackageClasses` element set to the service class `CatalogService.class`, which we shall develop later in this section. The package of each class specified in `basePackageClasses` is scanned. The modified class declaration for the `SpringMongoApplicationConfig` class is as follows.

```

@Configuration
@EnableMongoRepositories("com.mongo.repositories")
@ComponentScan(basePackageClasses = {CatalogService.class})
public class SpringMongoApplicationConfig extends AbstractMongoConfiguration {

}

```

The central repository marker interface is `org.springframework.data.repository.Repository`, and it provides CRUD access on top of the entities. The entity class `Catalog` is defined earlier in the chapter. The generic interface for CRUD operations on a repository is `org.springframework.data.repository.CrudRepository`. The MongoDB server specific repository interface is `org.springframework.data.mongodb.repository.MongoRepository<T, ID extends Serializable>`, which extends the `CrudRepository` interface. The interface is parameterized over the domain type, which would be `Catalog` for the example, and `ID` type, which is `String` in the example. The `ID` extends the `java.io.Serializable` interface to be able to serialize the `ID` in the MongoDB server. Create an interface, `CatalogRepository`, which extends the parameterized type `MongoRepository<Catalog, String>`. The `CatalogRepository` represents the MongoDB specific repository interface to store entities of type `Catalog` and with `Id` of type `String` in MongoDB server.

```
package com.mongo.repositories;
import org.springframework.data.mongodb.repository.MongoRepository;

import com.mongo.model.Catalog;

public interface CatalogRepository extends MongoRepository<Catalog, String> {
}
```

Create a service class `CatalogService` in which to access the repository instance from the context as follows.

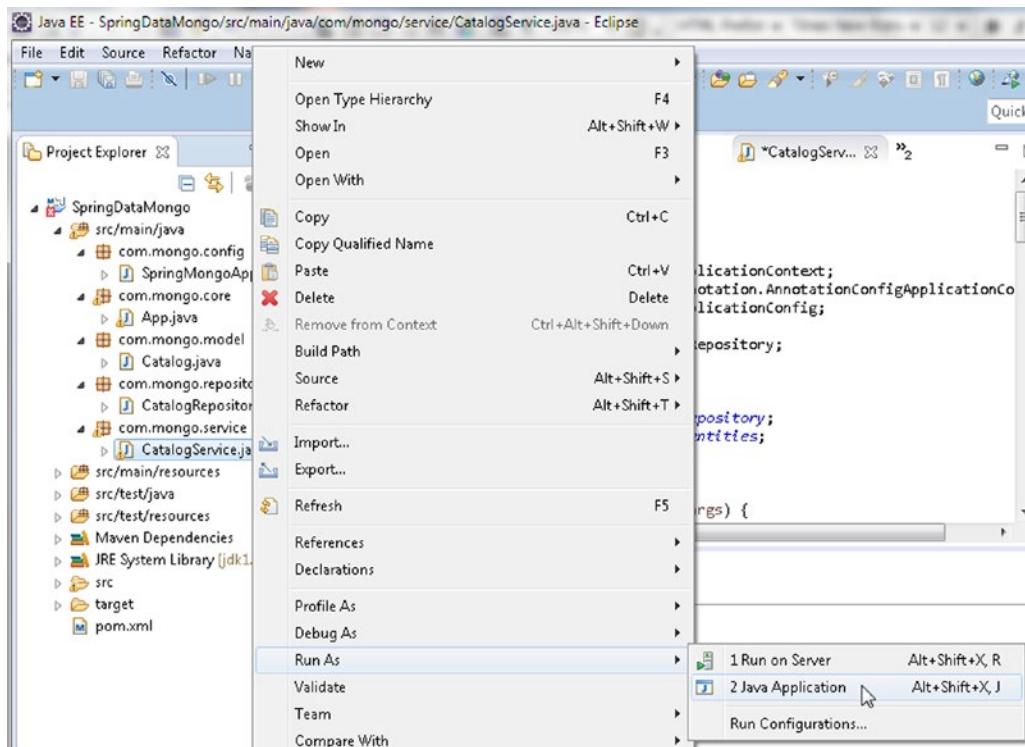
```
ApplicationContext context = new AnnotationConfigApplicationContext(
SpringMongoApplicationConfig.class);
CatalogRepository repository = context.getBean(CatalogRepository.class);
```

Subsequently, perform CRUD operations on the MongoDB document store using the `CatalogRepository` instance, which is declared as a class variable. Add the following (Table 10-17) methods to the `CatalogService` class for the CRUD operations. The methods names are same or similar to the `MongoRepository` methods' names.

**Table 10-17.** Methods in the `CatalogService` Class

| Method                          | Description                                       |
|---------------------------------|---------------------------------------------------|
| <code>count()</code>            | Counts the number of documents in the repository. |
| <code>findAll()</code>          | Finds all the documents in the repository.        |
| <code>save()</code>             | Saves a document to the repository.               |
| <code>saveBatch()</code>        | Saves a batch of documents to the repository.     |
| <code>findOne(String id)</code> | Finds a single document in the repository.        |
| <code>deleteAll()</code>        | Deletes all the documents in the repository.      |
| <code>deleteById()</code>       | Deletes a document by Id.                         |

Add method invocations for each of the methods in the `main` method. To run the `CatalogService` class right-click on the `CatalogService.java` application in the Package Explorer and select Run As ➤ Java Application as shown in Figure 10-22.



**Figure 10-22.** Running CatalogService Application

We shall invoke each method separately and comment out the other methods when the `CatalogService.java` application is run.

## Getting Document Count

The `MongoRepository<T, ID extends Serializable>` interface, which extends the `CrudRepository<T, ID>` interface and provides CRUD operation methods. The `MongoRepository` interface also includes a `count()` method, which returns the number of entities stored in a collection. To be able to count entities, first create some entities using the `MongoOperations` instance as discussed earlier in the chapter. Run the `App` application to invoke the `addDocumentBatch()` method to add some documents. Invoke the `count()` method using the `CatalogRepository` instance and output the long value returned.

```
public static void count() {
 System.out.println("Number of documents: " + repository.count());
}
```

The output in the Eclipse Console in Figure 10-23 shows that the number of document entities stored is 2.

The screenshot shows an IDE interface with several tabs at the top: Catalog.java, App.java, CatalogRepo..., \*CatalogServ... (highlighted), and »2. The CatalogService.java code is displayed in the main editor area:

```

12 public class CatalogService {
13
14 public static CatalogRepository repository;
15 public static ArrayList<Catalog> entities;
16 public static Catalog catalog1;
17 public static Catalog catalog2;
18
19 public static void main(String[] args) {
20
21 ApplicationContext context = new AnnotationConfigApplicationContext(
22 SpringMongoApplicationConfig.class);
23 repository = context.getBean(CatalogRepository.class);
24 // saveBatch();
25 count();
26 // findAll();
27 // save();
28 // saveBatch();
29
30

```

Below the editor is a terminal window titled "Console" showing the application's log output:

```

<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Number of documents: 2

```

**Figure 10-23.** Outputting Document Count

## Finding Entities from Repository

The MongoRepository interface provides the following (Table 10-18) methods for finding documents from repository.

**Table 10-18.** MongoRepository Methods for Finding Documents

| Method                                | Description                                                          |
|---------------------------------------|----------------------------------------------------------------------|
| T findOne(ID id)                      | Returns the entity instance for the specified ID.                    |
| Iterable<T> findAll()                 | Returns all entities of the entity type specified in the repository. |
| Iterable<T> findAll(Iterable<ID> ids) | Returns all entities of an entity type for the given IDs.            |
| findAll(Sort sort)                    | Returns all entities sorted by the given options.                    |

Before running the CatalogService application to find documents, do drop the catalog collection as we shall find the documents already in the catalog collection.

## Finding All Documents

To be able to find documents first create some entities using the MongoOperations instance as discussed earlier in the chapter. As an example, find all instances from the CatalogRepository instance repository using the `findAll()` method.

```
Iterable<Catalog> iterable = repository.findAll();
```

The `findAll()` method returns an `Iterable` from which obtain an `Iterator` using the `iterator()` method.

```
Iterator<Catalog> iter = iterable.iterator();
```

Iterate over the entity instances using the `Iterator` and output the fields for each entity instance. The `findAll()` method is as follows.

```
public static void findAll() {
 Iterable<Catalog> iterable = repository.findAll();
 Iterator<Catalog> iter = iterable.iterator();
 while (iter.hasNext()) {
 Catalog catalog = iter.next();

 System.out.println("Journal: " + catalog.getJournal());
 System.out.println("Publisher: " + catalog.getPublisher());
 System.out.println("Edition: " + catalog.getEdition());
 System.out.println("Title: " + catalog.getTitle());
 System.out.println("Author: " + catalog.getAuthor());
 }
}
```

When the CatalogService application is run the field values for the two entity instances are output in the Eclipse Console as shown in Figure 10-24.

The screenshot shows an IDE interface with several tabs at the top: Catalog.java, App.java, CatalogRepo..., CatalogServic..., and a tab with two small icons. Below the tabs is a code editor containing Java code for finding all documents in a Catalog repository. The code uses an Iterable to iterate over Catalog objects and prints their fields to System.out.

```

45
46 public static void findAll() {
47
48 Iterable<Catalog> iterable = repository.findAll();
49
50 Iterator<Catalog> iter = iterable.iterator();
51 while (iter.hasNext()) {
52 Catalog catalog = iter.next();
53
54 System.out.println("Journal: " + catalog.getJournal());
55 System.out.println("Publisher: " + catalog.getPublisher());
56 System.out.println("Edition: " + catalog.getEdition());
57 System.out.println("Title: " + catalog.getTitle());
58 System.out.println("Author: " + catalog.getAuthor());
59 }
60 }
61

```

Below the code editor is a terminal window titled "Console". It shows the application's startup log and the output of the findAll() method. The log includes SLF4J logger configuration messages. The output lists two Catalog objects with their respective fields: Journal, Publisher, Edition, Title, and Author.

```

<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert

```

**Figure 10-24.** Finding All Documents

## Finding One Document

As an example of using the `findOne()` method in `MongoRepository`, find the entity instance for a specific id, which is known to be in the MongoDB database. Subsequently output the field values for the entity instance. The `findOne()` method in the `CatalogService` class is as follows.

```

public static void findOne(String id) {
 Catalog catalog = repository.findOne(id);
 System.out.println("Journal: " + catalog.getJournal());
 System.out.println("Publisher: " + catalog.getPublisher());
 System.out.println("Edition: " + catalog.getEdition());
 System.out.println("Title: " + catalog.getTitle());
 System.out.println("Author: " + catalog.getAuthor());
}

```

Invoke the `findOne()` method from the `main` method using a specific id, which was save earlier.

```
if (repository.exists("53ea75e336845ce83eeb214f")) {
 findOne("53ea75e336845ce83eeb214f");
}
```

The field values for the entity instance are output in the Eclipse Console as shown in Figure 10-25.

The screenshot shows the Eclipse IDE interface. At the top, there are tabs for Catalog.java, App.java, CatalogRepo..., and CatalogServic... (which is currently selected). Below the tabs, the Catalog.java code is displayed:

```
if (repository.exists("55cb4f50db9499f5a242ad93")) {
 findOne("55cb4f50db9499f5a242ad93");
}

public static void findOne(String id) {
 Catalog catalog = repository.findOne(id);
 System.out.println("Journal: " + catalog.getJournal());
 System.out.println("Publisher: " + catalog.getPublisher());
 System.out.println("Edition: " + catalog.getEdition());
 System.out.println("Title: " + catalog.getTitle());
 System.out.println("Author: " + catalog.getAuthor());
}
```

Below the code editor is the Eclipse Console window, which displays the application's output:

```
<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
```

**Figure 10-25.** Finding One Document

## Saving Entities

The `MongoRepository` interface provides the following methods (Table 10-19) for saving documents using the repository.

**Table 10-19.** MongoRepository Methods for Saving Documents

| Method                                               | Description                                                                                                                             |
|------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <S extends T> S save(S entity)                       | Saves a given entity. If the entity is already in the server, overwrites the entity. Returns the saved entity instance.                 |
| <S extends T> Iterable<S> save(Iterable<S> entities) | Saves a given collection of entities. If an entity is already in the server, overwrites the entity. Returns the saved entity instances. |

## Saving a Single Document

As an example, create and save an entity instance using the `save(S entity)` method.

1. In the `save()` method in the `CatalogService` class create a `Catalog` instance. The `id` field argument may be an empty String as the `_id` field is generated automatically.
2. Subsequently, invoke the `save()` method using the `Catalog` instance as method argument. To find the saved documents invoke the `findAll()` method in `CatalogService`. The `save()` method in `CatalogService` is as follows.

```
public static void save() {
 Catalog catalog = new Catalog("", "Oracle Magazine",
 "Oracle Publishing", "11-12-2013", "Engineering as a Service",
 "Kelly, David");
 repository.save(catalog);
 findAll();
}
```

3. Invoke the `save()` method from the main method to save a `Catalog` instance and subsequently find and list the document's field values as shown in Figure 10-26.

The screenshot shows an IDE interface with several tabs at the top: Catalog.java, App.java, CatalogRepo..., CatalogServic..., and CatalogService. The CatalogService tab is active, displaying the following Java code:

```
35 save();
36 }
37
38 public static void save() {
39
40 Catalog catalog = new Catalog("", "Oracle Magazine",
41 "Oracle Publishing", "11-12-2013", "Engineering as a Service",
42 "Kelly, David");
43 repository.save(catalog);
44 findAll();
45 }
46
47 }
```

Below the code editor is a toolbar with icons for copy, cut, paste, and others. The bottom part of the interface is the 'Console' window, which displays the application's output:

```
<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: 11-12-2013
Title: Engineering as a Service
Author: Kelly, David
```

**Figure 10-26.** Saving One Document

# Saving a Batch of Documents

As an example of using the `save(Iterable<S> entities)` method, create an `ArrayList` of entity instances in the `saveBatch()` method in `CatalogService` application. Invoke the `save(Iterable<S> entities)` method with the `ArrayList` instance as an argument.

```
repository.save(arrayList);
```

Subsequently invoke the `findAll()` method to find all the saved entities. The `saveBatch()` method is as follows.

```
public static void saveBatch() {

 Catalog catalog1 = new Catalog("", "Oracle Magazine",
 "Oracle Publishing", "11-12-2013", "Engineering as a Service",
 "Kelly, David");

 Catalog catalog2 = new Catalog("", "Oracle Magazine",
 "Oracle Publishing", "11-12-2013",
 "Quintessential and Collaborative", "Haunert, Tom");
 entities = new ArrayList<Catalog>();
```

```

entities.add(catalog1);
entities.add(catalog2);
repository.save(entities);
findAll();
}

}

```

Invoke the `saveBatch()` method from the `main` method. Before running the `CatalogService` application drop the previously created `catalog` collection using the `db.catalog.drop()` method in the Mongo shell. When the `CatalogService` application is run the collection of entity instances in the `ArrayList` get added to the MongoDB server. The saved `Catalog` entity field values get output to the Eclipse Console as shown in Figure 10-27.

The screenshot shows the Eclipse IDE interface with several open tabs at the top: Catalog.java, App.java, CatalogRepo..., CatalogService..., and CatalogService... (with a double arrow icon). The CatalogService... tab is active. Below the tabs, the code for CatalogService.java is displayed:

```

Catalog.java App.java CatalogRepo... CatalogService... »2
37 saveBatch();
38 }
39
40 public static void saveBatch() {
41
42 catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
43 "11-12-2013", "Engineering as a Service", "Kelly, David");
44
45 catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
46 "11-12-2013", "Quintessential and Collaborative",
47 "Haunert, Tom");
48 entities = new ArrayList<Catalog>();
49 entities.add(catalog1);
50 entities.add(catalog2);
51 repository.save(entities);
52 findAll();
53
54 }

```

Below the code editor is the Eclipse Console window, which displays the application's output:

```

Console X
<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: 11-12-2013
Title: Engineering as a Service
Author: Kelly, David
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: 11-12-2013
Title: Quintessential and Collaborative
Author: Haunert, Tom

```

**Figure 10-27.** Saving a Batch of Documents

## Deleting Entities

The MongoRepository interface provides the following (Table 10-20) methods for deleting documents using the repository.

**Table 10-20.** MongoRepository Methods for Deleting Documents

| Method                                                         | Description                                                               |
|----------------------------------------------------------------|---------------------------------------------------------------------------|
| <code>void delete(ID id)</code>                                | Deletes the entity by the given ID managed by the repository.             |
| <code>void delete(T entity)</code>                             | Deletes the specified entity managed by the repository.                   |
| <code>void delete(Iterable&lt;? extends T&gt; entities)</code> | Deletes the entities in the specified Iterable managed by the repository. |
| <code>void deleteAll()</code>                                  | Deletes all entities managed by the repository.                           |

## Deleting a Document By Id

As an example of using the `delete(ID id)` method delete the entity with a known ID that is in the MongoDB server in the `deleteById()` method in CatalogService application.

First, save some documents in the `deleteById()` method. Use class variables `catalog1` and `catalog2` for the Catalog entities saved. In the `deleteById()` method invoke the `delete(ID id)` method of the `CatalogRepository` instance with the Id for one of the saved documents as method argument. The id for a document is obtained using the `getId()` method. Subsequently invoke the `findAll()` method to find all document.

```
public static void deleteById() {
 //Add documents
 repository.delete(catalog1.getId());
 findAll();
}
```

In the `main` method of `CatalogService` invoke the `deleteById()` method.

```
deleteById();
```

When the `CatalogService` application is run the `Catalog` instance for the specified Id gets deleted. The subsequent call to `findAll()` lists only one of the saved documents as shown in Figure 10-28.

The screenshot shows an IDE interface with several tabs at the top: Catalog.java, App.java, CatalogRepo..., CatalogServic..., and CatalogService... (active tab). The CatalogService... tab contains Java code for deleting documents by ID. The code creates two Catalog objects, adds them to a list, saves the list to a repository, and then deletes the first document by its ID. Below the code editor is a terminal window titled 'Console' showing the application's log output. The log includes SLF4J messages about failed class loading and defaulting to NOP logger, followed by the details of the two catalog entries: Oracle Magazine, Oracle Publishing, 11-12-2013, Engineering as a Service, Kelly, David; and Quintessential and Collaborative, Haunert, Tom.

```

Catalog.java App.java CatalogRepo... CatalogService... »2
58
59 deleteById();
60 }
61
62 public static void deleteById() {
63 catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
64 "11-12-2013", "Engineering as a Service", "Kelly, David");
65
66 catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
67 "11-12-2013", "Quintessential and Collaborative",
68 "Haunert, Tom");
69 entities = new ArrayList<Catalog>();
70 entities.add(catalog1);
71 entities.add(catalog2);
72 repository.save(entities);
73 repository.delete(catalog1.getId());
74 findAll();
75 }
76

```

```

Console X
<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: 11-12-2013
Title: Quintessential and Collaborative
Author: Haunert, Tom

```

**Figure 10-28.** Deleting a Document By Id

## Deleting All Documents

As an example of using the `deleteAll()` method, first add a batch of documents in the `deleteAll()` method in `CatalogService`. Subsequently invoke the `deleteAll()` method of the `CatalogRepository` instance. Subsequently invoke the `findAll()` method to find all the documents.

```

public static void deleteAll() {
 //Add documents
 repository.deleteAll();
 findAll();
}

```

In the `main` method of `CatalogService` invoke the `deleteAll()` method.

```
deleteAll();
```

When the CatalogService application is run a batch of documents are saved and subsequently the deleteAll() method deletes all the documents saved. The subsequent invocation of the findAll() method does not find and list any documents as all documents have been deleted as shown in Figure 10-29.

The screenshot shows an IDE interface with several tabs at the top: Catalog.java, App.java, CatalogRepo..., CatalogServic... (highlighted), and a file icon with '2'. Below the tabs is the code for CatalogService.java:

```

40
41 deleteAll();
42 }
43
44 public static void deleteAll() {
45 catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
46 "11-12-2013", "Engineering as a Service", "Kelly, David");
47
48 catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
49 "11-12-2013", "Quintessential and Collaborative",
50 "Haunert, Tom");
51 entities = new ArrayList<Catalog>();
52 entities.add(catalog1);
53 entities.add(catalog2);
54 repository.save(entities);
55 repository.deleteAll();
56 findAll();
57 }

```

Below the code editor is a terminal window titled 'Console' showing the application's log output:

```

<terminated> CatalogService [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Aug 12, 2015,
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.

```

**Figure 10-29.** Deleting All Documents

The CatalogService class is listed:

```

package com.mongo.service;

import java.util.ArrayList;
import java.util.Iterator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.mongo.config.SpringMongoApplicationConfig;
import com.mongo.model.Catalog;
import com.mongo.repositories.CatalogRepository;

public class CatalogService {

 public static CatalogRepository repository;
 public static ArrayList<Catalog> entities;
}

```

```
public static Catalog catalog1;
public static Catalog catalog2;

public static void main(String[] args) {

 ApplicationContext context = new AnnotationConfigApplicationContext(
 SpringMongoApplicationConfig.class);
 repository = context.getBean(CatalogRepository.class);
 // saveBatch();
 // count();
 // findAll();
 // saveBatch();
 // deleteAll();
 // deleteById();

 /*
 * if (repository.exists("55cb4f50db9499f5a242ad93")) {
 * findOne("55cb4f50db9499f5a242ad93"); }
 */
 // save();

 // saveBatch();

 // deleteById();

 deleteAll();
}

public static void deleteAll() {
 catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
 "11-12-2013", "Engineering as a Service", "Kelly, David");

 catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
 "11-12-2013", "Quintessential and Collaborative",
 "Haunert, Tom");
 entities = new ArrayList<Catalog>();
 entities.add(catalog1);
 entities.add(catalog2);
 repository.save(entities);
 repository.deleteAll();
 findAll();
}

public static void deleteById() {
 catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
 "11-12-2013", "Engineering as a Service", "Kelly, David");

 catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
 "11-12-2013", "Quintessential and Collaborative",
 "Haunert, Tom");
 entities = new ArrayList<Catalog>();
```

```
entities.add(catalog1);
entities.add(catalog2);
repository.save(entities);
repository.delete(catalog1.getId());
findAll();
}

public static void saveBatch() {

catalog1 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
"11-12-2013", "Engineering as a Service", "Kelly, David");

catalog2 = new Catalog("", "Oracle Magazine", "Oracle Publishing",
"11-12-2013", "Quintessential and Collaborative",
"Haunert, Tom");
entities = new ArrayList<Catalog>();
entities.add(catalog1);
entities.add(catalog2);
repository.save(entities);
findAll();
}

public static void save() {

Catalog catalog = new Catalog("", "Oracle Magazine",
"Oracle Publishing", "11-12-2013", "Engineering as a Service",
"Kelly, David");
repository.save(catalog);
findAll();
}

public static void findOne(String id) {

Catalog catalog = repository.findOne(id);
System.out.println("Journal: " + catalog.getJournal());
System.out.println("Publisher: " + catalog.getPublisher());
System.out.println("Edition: " + catalog.getEdition());
System.out.println("Title: " + catalog.getTitle());
System.out.println("Author: " + catalog.getAuthor());
}

public static void count() {
System.out.println("Number of documents: " + repository.count());
}

public static void findAll() {
```

```
Iterable<Catalog> iterable = repository.findAll();

Iterator<Catalog> iter = iterable.iterator();
while (iter.hasNext()) {
 Catalog catalog = iter.next();

 System.out.println("Journal: " + catalog.getJournal());
 System.out.println("Publisher: " + catalog.getPublisher());
 System.out.println("Edition: " + catalog.getEdition());
 System.out.println("Title: " + catalog.getTitle());
 System.out.println("Author: " + catalog.getAuthor());
}
}
```

## Summary

In this chapter we used the Spring Data MongoDB project with MongoDB server. The common CRUD operations on a MongoDB datastore may be performed using the `org.springframework.data.mongodb.core.MongoOperations` interface and the `org.springframework.data.mongodb.repository.MongoRepository`. In this chapter we discussed CRUD operations on MongoDB using these two interfaces. In the next chapter we shall discuss using MongoDB with Hadoop and create an Apache Hive table over MongoDB.

## CHAPTER 11



# Creating an Apache Hive Table with MongoDB

MongoDB is the leading NoSQL database. MongoDB is based on the BSON (Binary JSON) JSON-style document format, which is based on dynamic schemas providing flexibility in storage. The Hive MongoDB Storage Handler makes it feasible to access MongoDB from Apache Hive. A Hive external table may be created on a MongoDB document store. In this chapter we shall create a document store in MongoDB, and subsequently create a Hive external table on the MongoDB document store.

Apache Hadoop is a framework for distributed processing of large data sets. Apache Hadoop file system is the Hadoop Distributed File System (HDFS). Apache Hive is a software for querying and managing large data sets stored on a distributed storage, which is the HDFS by default. This chapter assumes knowledge of Apache Hadoop (Reference: <https://hadoop.apache.org/>) and Apache Hive (Reference: <https://hive.apache.org/>) as we won't be discussing what is HDFS or how HDFS stores data for Hive. We shall also make use of some Hadoop shell commands. This chapter includes the following topics:

- Overview of Hive storage handler for MongoDB
- Setting up the environment
- Creating a MongoDB datastore
- Creating an external table in Hive

## Overview of Hive Storage Handler for MongoDB

Apache Hive supports two types of tables: *managed* tables and *external* tables. The difference between the two is that a managed table is managed by Hive, which implies that both the metadata and the data for a Hive managed table are managed by Hive. For an external table, Hive only manages the metadata – not the table data. If a Hive managed table is dropped, both the data and the metadata get dropped. But if a Hive external table is dropped, only the metadata is dropped and not the table data. A Hive external table is suitable if the data is stored in an external datasource such as MongoDB database. In this section we introduce the Hive MongoDB Storage Handler, which is used to create a Hive external table over a MongoDB database.

The MongoDB storage handler for Hive class is `org.yong3.hive.mongo.MongoStorageHandler`. The storage handler supports only the Hive primitive types such as `int` and `string`. The MongoDB storage handler provides serde properties `mongo.column.mapping` in which the MongoDB datastore column names that are to be mapped to the Hive external table are specified. In addition the following (Table 11-1) `tblproperties` are supported.

**Table 11-1.** Mongo Storage Handler Properties

| Property         | Description         |
|------------------|---------------------|
| mongo.host       | MongoDB host name.  |
| mongo.port       | MongoDB port.       |
| mongo.db         | MongoDB database.   |
| mongo.user       | MongoDB user name.  |
| mongo.passwd     | MongoDB password.   |
| mongo.collection | MongoDB collection. |

## Setting Up the Environment

The following software is required for the chapter.

- Hadoop 2.0.0 CDH 4.6
- Hive 0.10.0 CDH 4.6
- MongoDB Java Driver 2.11.3
- MongoDB Storage Handler for Hive 0.0.3
- MongoDB 2.6.3
- Eclipse IDE for Java EE Developers
- Java 7

Later versions of the listed software may also be used. Download the MongoDB storage handler for Hive from <https://github.com/yc-huang/Hive-mongo> and extract the zip file to a directory. The Jar files with the storage handler are in the `Hive-mongo-master/release` directory. Use the `hive-mongo-0.0.3-jar-with-dependencies.jar` file, which has all the required dependencies included. Download the MongoDB Java driver Jar file `mongo-java-driver-2.11.3.jar` or a later version from <http://central.maven.org/maven2/org/mongodb/mongo-java-driver/>.

Complete the following steps to set up the environment:

1. We have used Oracle Linux 6.5, which is installed on Oracle VirtualBox 4.3. But a different distribution of Linux may be used instead. Oracle Linux is based on RedHat Linux, one of the most commonly used Linux distributions. Create a directory for MongoDB and other software and set its permissions.

```
mkdir /mongodb
chmod -R 777 /mongodb
cd /mongodb
```

2. Download Java 7 and extract the file to the `/mongodb` directory.

```
tar zxvf jdk-7u55-linux-i586.gz
```

3. Download Hadoop 2.0.0 and extract the tar.gz file to a directory.

```
wget http://archive.cloudera.com/cdh4/cdh/4/hadoop-2.0.0-cdh4.6.0.tar.gz
tar -xvf hadoop-2.0.0-cdh4.6.0.tar.gz
```

4. Create symlinks for the Hadoop bin and conf directories.

```
ln -s /mongodb/hadoop-2.0.0-cdh4.6.0/bin /oranosql/hadoop-2.0.0-cdh4.6.0/share/
hadoop/mapreduce2/bin
ln -s /mongodb/hadoop-2.0.0-cdh4.6.0/etc/hadoop /oranosql/hadoop-2.0.0-cdh4.6.0/
share/hadoop/mapreduce2/conf
```

5. Configure Hadoop in the core-site.xml and hdfs-site.xml configuration files. In the core-site.xml, which is listed below, set the fs.defaultFS and hadoop.tmp.dir properties.

```
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
 <name>fs.defaultFS</name>
 <value>hdfs://10.0.2.15:8020</value>
</property>
<property>
 <name>hadoop.tmp.dir</name>
 <value>file:///var/lib/hadoop-0.20/cache</value>
</property>
</configuration>
```

6. Create the directory specified for the hadoop.tmp.dir property and set its permissions to global (777).

```
mkdir -p /var/lib/hadoop-0.20/cache
chmod -R 777 /var/lib/hadoop-0.20/cache
```

7. In the hdfs-site.xml configuration file, which is listed below, set the dfs.permissions.superusergroup, dfs.namenode.name.dir, dfs.replication, and dfs.permissions properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
 <name>dfs.permissions.superusergroup</name>
 <value>hadoop</value>
</property><property>
 <name>dfs.namenode.name.dir</name>
 <value>file:///data/1/dfs/nn</value>
</property>
<property>
```

```

<name>dfs.replication</name>
 <value>1</value>
</property>
<property>
<name>dfs.permissions</name>
<value>false</value>
</property>
</configuration>
```

8. Create the NameNode storage directory and set its permissions.

```
mkdir -p /data/1/dfs/nn
chmod -R 777 /data/1/dfs/nn
```

9. Download and install Hive 0.10.0 CDH 4.6.

```
wget http://archive.cloudera.com/cdh4/cdh/4/hive-0.10.0-cdh4.6.0.tar.gz
tar -xvf hive-0.10.0-cdh4.6.0.tar.gz
```

10. Create the `hive-site.xml` file from the template.

```
cd /mongodb/hive-0.10.0-cdh4.6.0/conf
cp hive-default.xml.template hive-site.xml
```

11. By default, Hive makes use of the Embedded metastore. We shall use the Remote metastore for which we need to configure the `hive.metastore.uris` property to the remote metastore URI. Also set the `hive.metastore.warehouse.dir` property to the Hive storage directory, the directory in which Hive databases and tables are stored. The `hive-site.xml` configuration file is listed:

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
<name>hive.metastore.warehouse.dir</name>
<value>hdfs://10.0.2.15:8020/user/hive/warehouse</value>
</property>
</configuration>
<property>
<name>hive.metastore.uris</name>
<value>thrift://localhost:10000</value>
</property>
</configuration>
```

12. Create the HDFS path directory specified in the `hive.metastore.warehouse.dir` property and set its permissions.

```
hadoop dfs -mkdir hdfs://10.0.2.15:8020/user/hive/warehouse
hadoop dfs -chmod -R g+w hdfs://10.0.2.15:8020/user/hive/warehouse
```

13. Download and extract the MongoDB 2.6.3 file.

```
curl -O http://downloads.mongodb.org/linux/mongodb-linux-i686-2.6.3.tgz
tar -zvxf mongodb-linux-i686-2.6.3.tgz
```

14. Copy the mongo-java-driver-2.6.3.jar and hive-mongo-0.0.3-jar-with-dependencies.jar to the /mongodb/hive-0.10.0-cdh4.6.0/lib directory. Set the environment variables for Hadoop, Hive, Java, and MongoDB in the bash shell.

```
vi ~/.bashrc
export HADOOP_PREFIX=/mongodb/hadoop-2.0.0-cdh4.6.0
export HADOOP_CONF=$HADOOP_PREFIX/etc/hadoop
export MONGO_HOME=/mongodb/mongodb-linux-i686-2.6.3
export HIVE_HOME=/mongodb/hive-0.10.0-cdh4.6.0
export HIVE_CONF=$HIVE_HOME/conf
export JAVA_HOME=/mongodb/jdk1.7.0_55
export HADOOP_MAPRED_HOME=/mongodb/hadoop-2.0.0-cdh4.6.0/bin
export HADOOP_HOME=/mongodb/hadoop-2.0.0-cdh4.6.0/share/hadoop/mapreduce2
export HADOOP_CLASSPATH=$HADOOP_HOME/*:$HADOOP_HOME/lib/*:$HIVE_HOME/lib/*:/mongodb/mongo-java-driver-2.6.3.jar:/mongodb/hive-mongo-0.0.3-jar-with-dependencies.jar:$HIVE_CONF
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_MAPRED_HOME::$HIVE_HOME/bin:$MONGO_HOME/bin
```

15. Format the NameNode and start the HDFS, which comprises the NameNode and DataNode.

```
hadoop namenode -format
hadoop namenode
hadoop datanode
```

16. Create a directory (and set its permissions) in the HDFS to put the Hive directory to make Hive available in the runtime classpath.

```
hdfs dfs -mkdir hdfs://localhost:8020/mongodb
hadoop dfs -chmod -R g+w hdfs://localhost:8020/mongodb
```

17. Put the Hive directory in HDFS.

```
hdfs dfs -put /mongodb/hive-0.10.0-cdh4.6.0 hdfs://localhost:8020/mongodb
```

## Creating a MongoDB Data Store

In this section we shall create a MongoDB datastore. We shall store the following sample log (taken from WebLogic Server) in MongoDB.

```
Apr-8-2014-7:06:16-PM-PDT Notice WebLogicServer AdminServer BEA-000365 Server state changed to STANDBY
Apr-8-2014-7:06:17-PM-PDT Notice WebLogicServer AdminServer BEA-000365 Server state changed to STARTING
Apr-8-2014-7:06:18-PM-PDT Notice WebLogicServer AdminServer BEA-000360 Server started in RUNNING mode
```

1. Start MongoDB server to be able to access MongoDB using the following command.

```
mongod
```

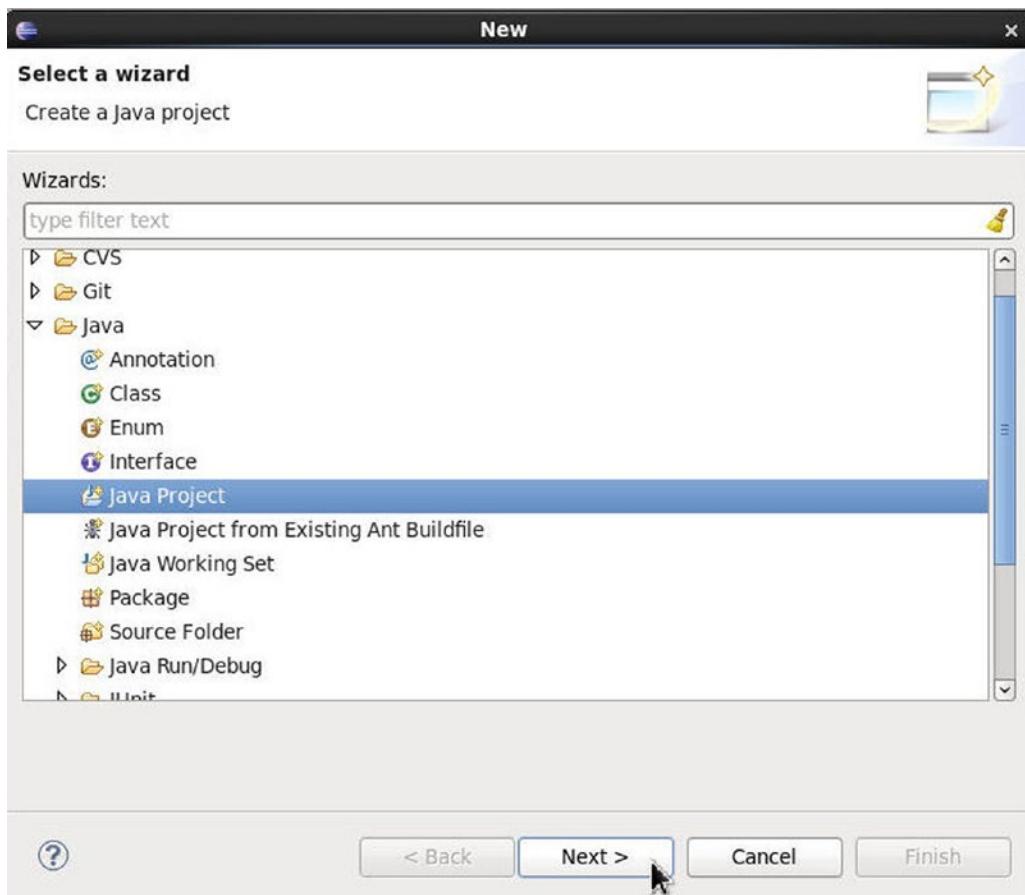
MongoDB gets started. The output from the `mongod` command is listed:

```
[root@localhost mongodb]# mongod
mongod --help for help and startup options
2014-06-22T12:18:56.908-0400
2014-06-22T12:18:56.918-0400 warning: 32-bit servers don't have journaling
enabled by default. Please use --journal if you want durability.
2014-06-22T12:18:56.920-0400
2014-06-22T12:18:57.112-0400 [initandlisten] MongoDB starting : pid=2610
port=27017 dbpath=/data/db 32-bit host=localhost.oraclelinux
2014-06-22T12:18:57.117-0400 [initandlisten]
2014-06-22T12:18:57.119-0400 [initandlisten] ** NOTE: This is a 32 bit MongoDB
binary.
2014-06-22T12:18:57.124-0400 [initandlisten] ** 32 bit builds are limited
to less than 2GB of data (or less with --journal).
2014-06-22T12:18:57.129-0400 [initandlisten] ** Note that journaling
defaults to off for 32 bit and is currently off.
2014-06-22T12:18:57.130-0400 [initandlisten] ** See http://dochub.mongodb.org/core/32bit
2014-06-22T12:18:57.136-0400 [initandlisten]
2014-06-22T12:18:57.153-0400 [initandlisten] db version v2.6.3
2014-06-22T12:18:57.167-0400 [initandlisten] git version:
255f67a66f9603c59380b2a389e386910bbb52cb
2014-06-22T12:18:57.203-0400 [initandlisten] build info: Linux ip-10-225-17-
11 2.6.18-194.32.1.el5xen #1 SMP Mon Dec 20 11:08:09 EST 2010 i686 BOOST_LIB_
VERSION=1_49
2014-06-22T12:18:57.206-0400 [initandlisten] allocator: system
2014-06-22T12:18:57.206-0400 [initandlisten] options: {}
2014-06-22T12:18:58.582-0400 [initandlisten] waiting for connections on port
27017
```

2. Next, we shall create a MongoDB data store using a Java application in Eclipse IDE. Start Eclipse IDE from the Eclipse installation directory.

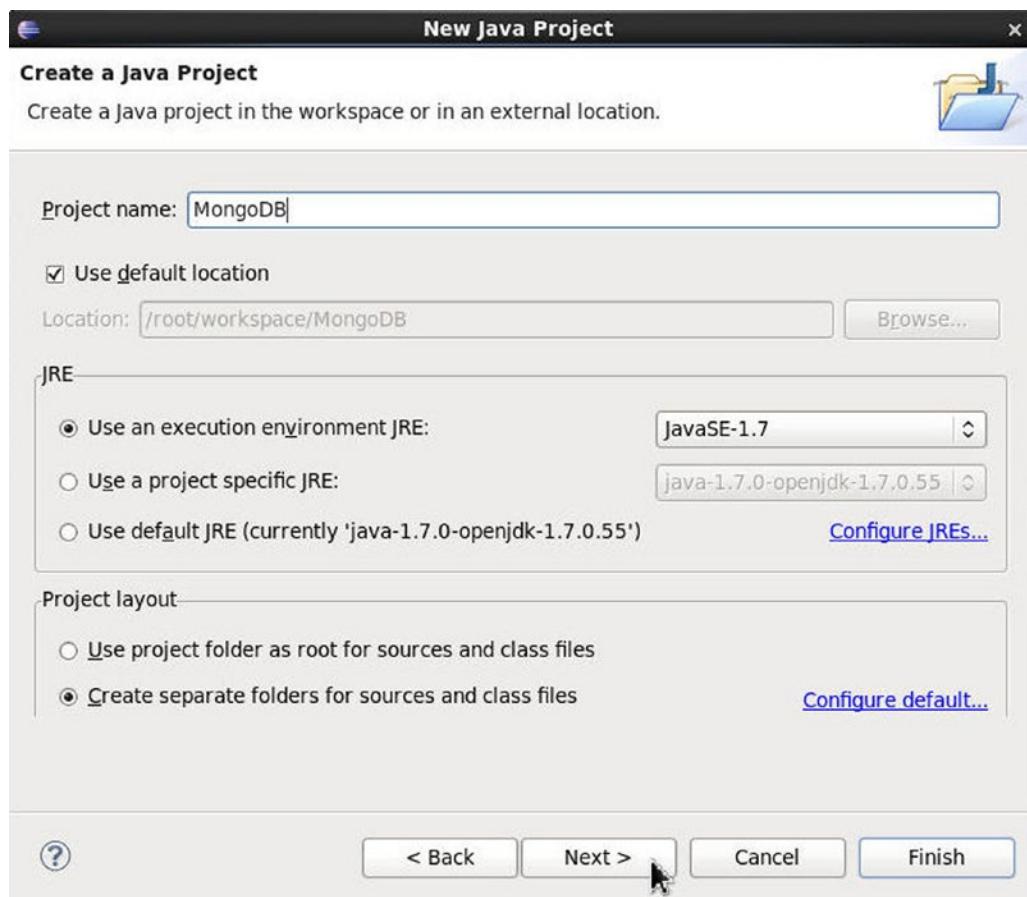
```
./eclipse
```

3. Click on File ► New. In the New window, select Java>Java Project and click on Next as shown in Figure 11-1.



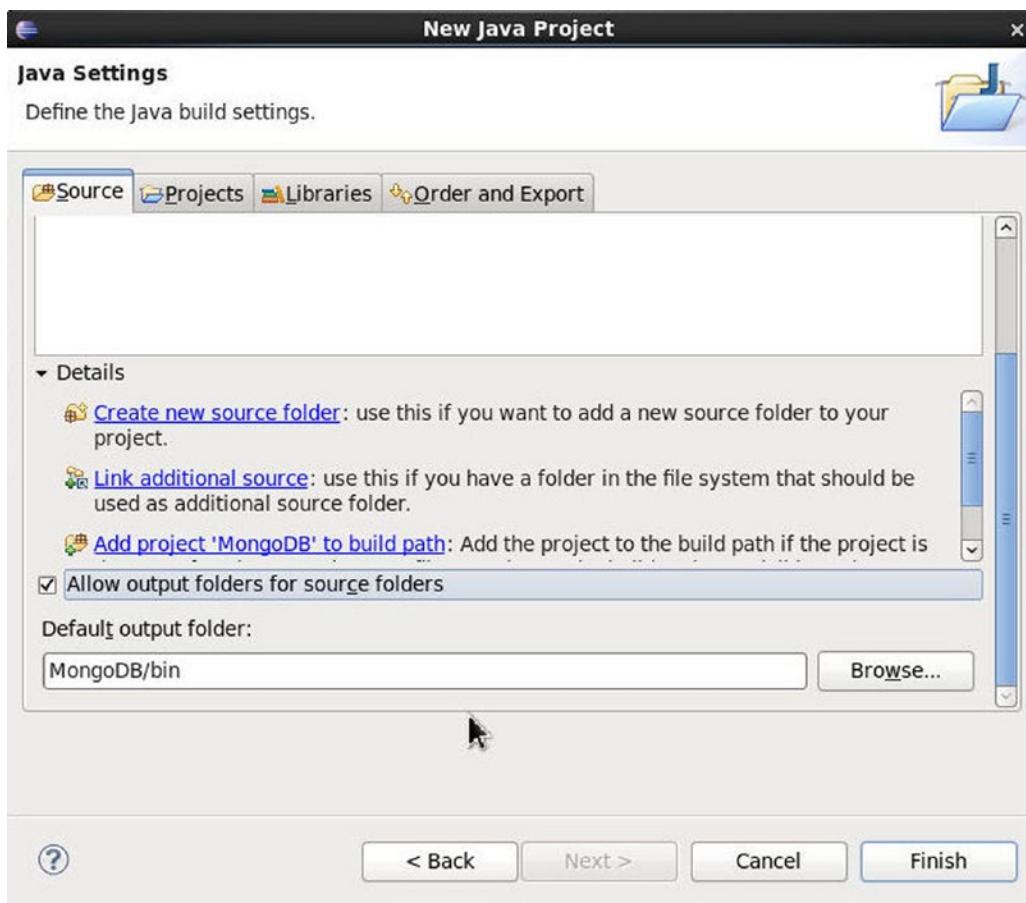
**Figure 11-1.** Selecting Java ► Java Project

4. In New Java Project specify a Project name (MongoDB), select Use default location, select a JRE, and click on Next as shown in Figure 11-2.



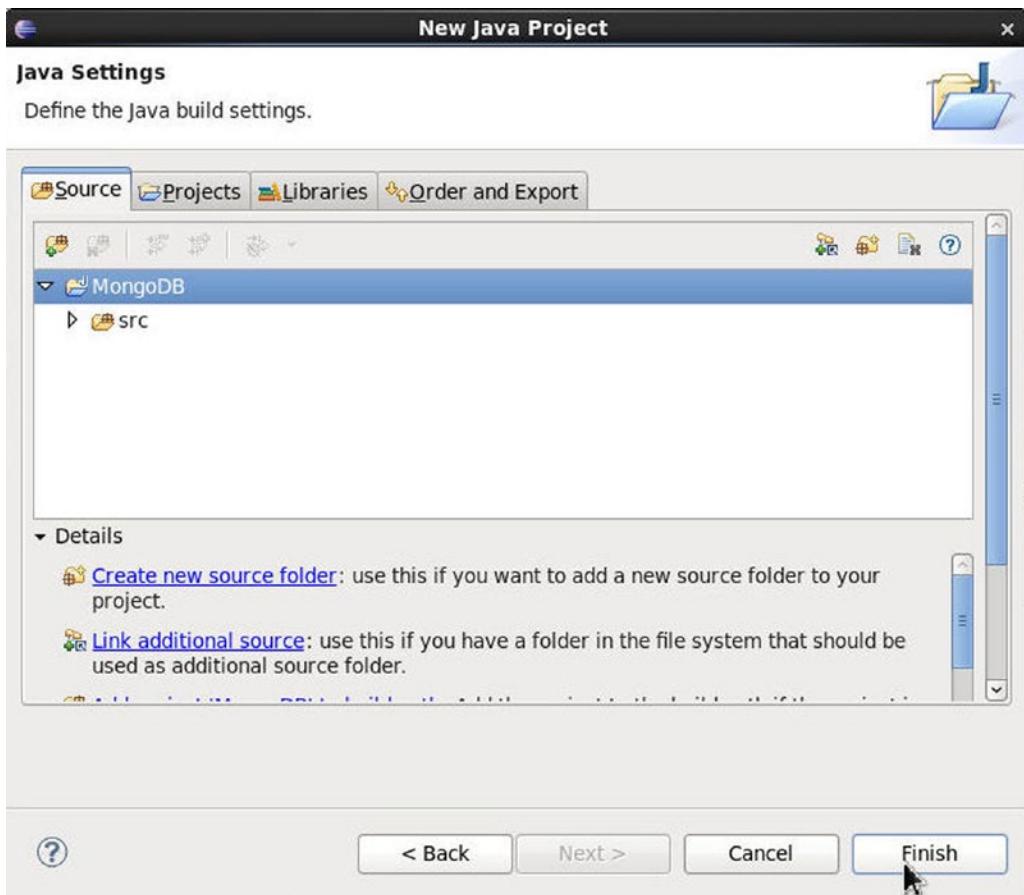
**Figure 11-2.** Creating a Java Project

5. In Java Settings select Allow output folders for source folders. The Default folder name is prespecified as MongoDB/bin as shown in Figure 11-3.



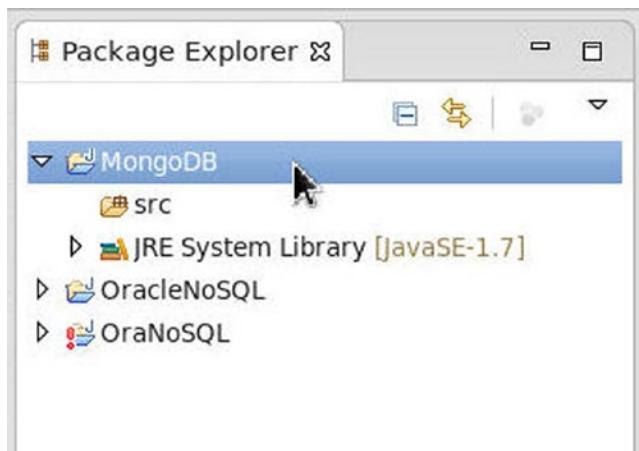
**Figure 11-3.** Configuring Output Folder

6. The Source folder is also preselected as MongoDB/src as shown in Figure 11-4.  
Click on Finish.



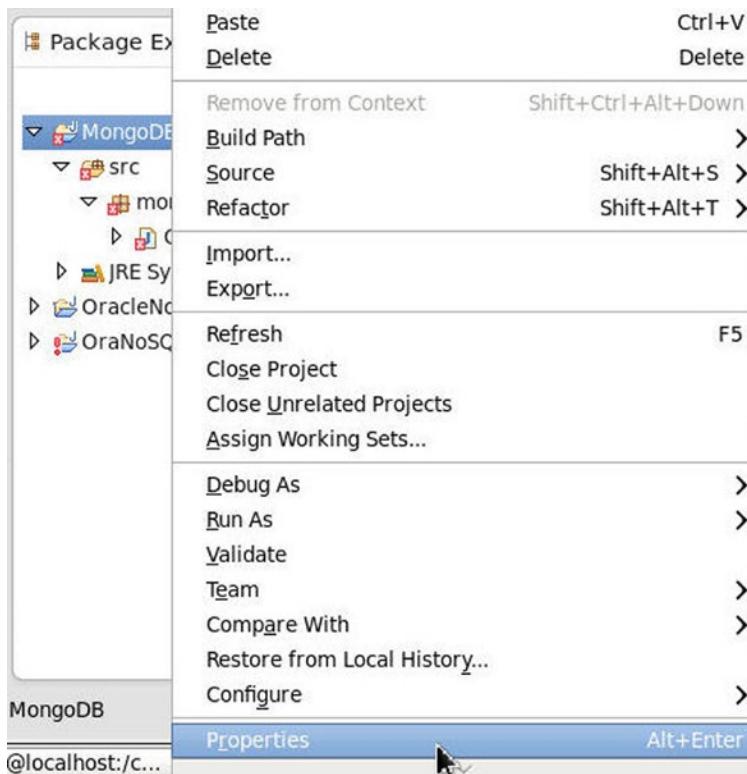
**Figure 11-4.** Configuring Source Folder

A Java project MongoDB gets created as shown in Figure 11-5.



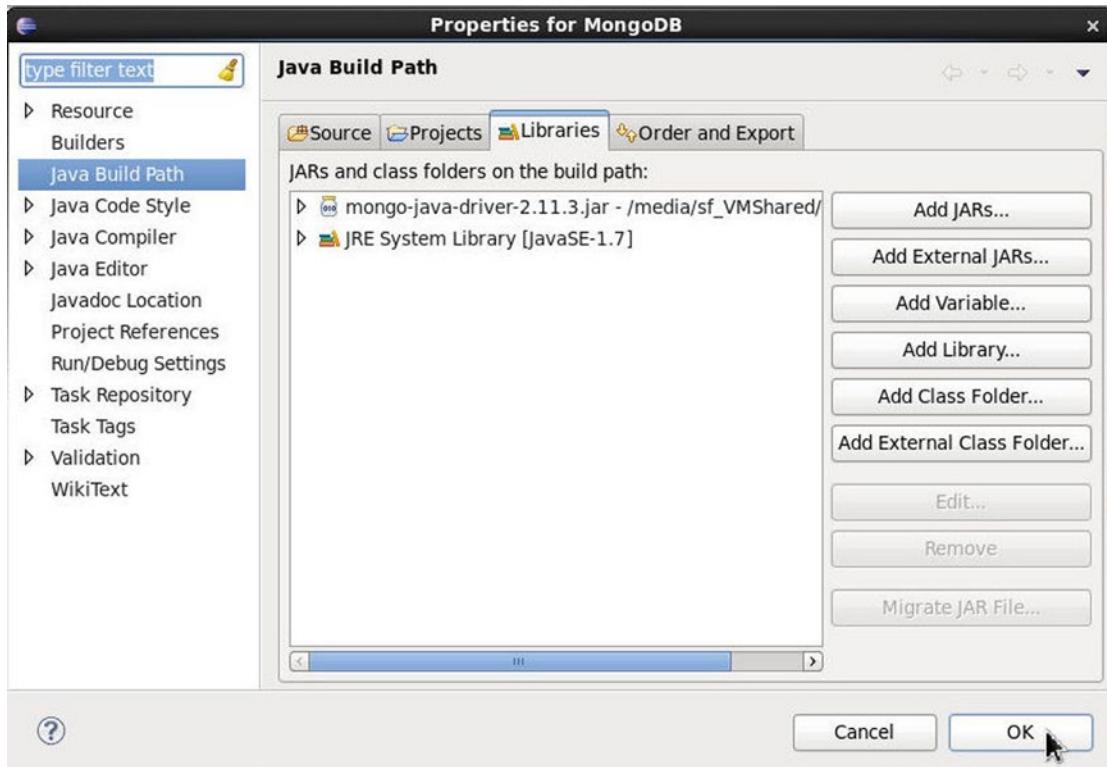
**Figure 11-5.** Java Project MongoDB

7. We need to add the MongoDB Java driver Jar file to the project Java Build Path. Right-click on MongoDB project node and select Properties as shown in Figure 11-6.



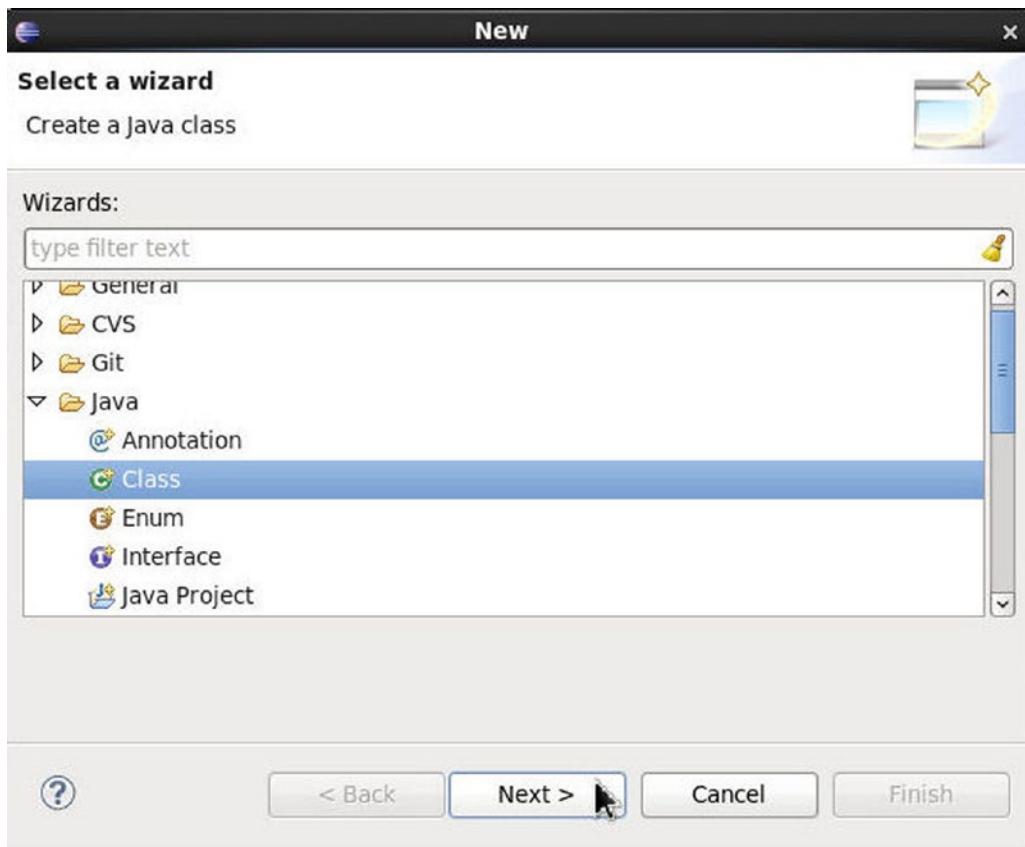
**Figure 11-6.** Selecting Properties for MongoDB Project

8. In Properties for MongoDB select Java Build Path and click on Add External JARs to add the MongoDB Java driver jar file mongo-java-driver-2.11.3.jar as shown in Figure 11-7. Click on OK.



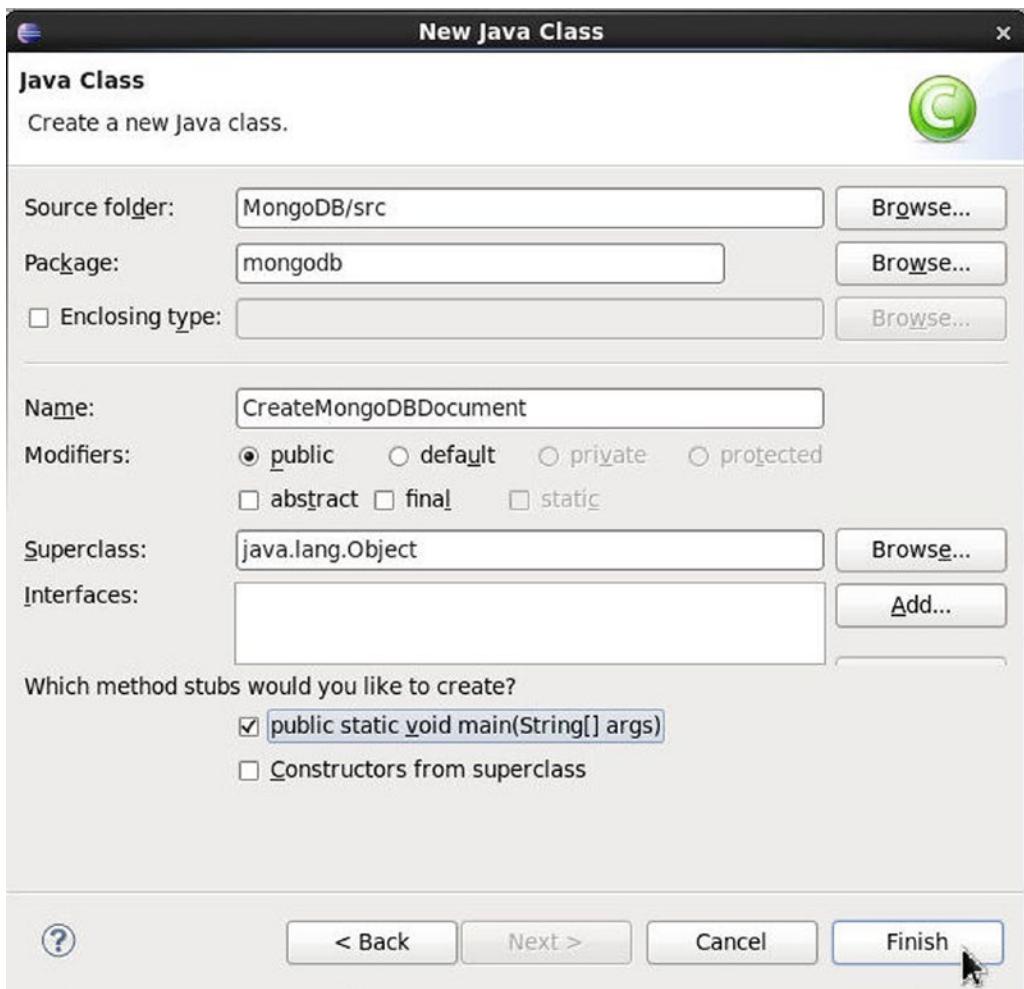
**Figure 11-7.** Adding MongoDB Java Driver to Classpath

9. Next, add a Java class to the Java project. Select File > New and in the New window select the Java > Class wizard and click on Next as shown in Figure 11-8.



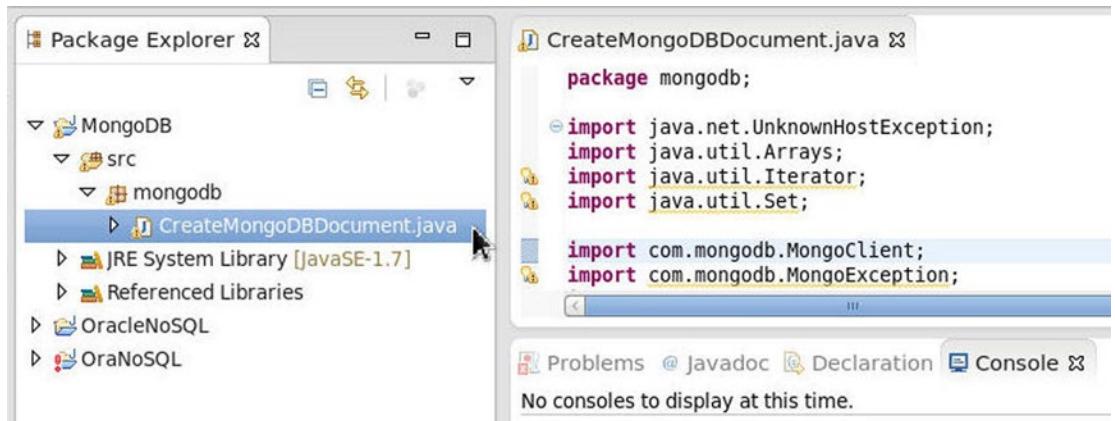
**Figure 11-8.** Selecting Java ► Class

10. In New Java Class the Source folder is prespecified as MongoDB/src. Specify a Package name, mongodb. Specify a class Name, CreateMongoDBDocument as shown in Figure 11-9. Select the main method stub to add to the class and click on Finish.



**Figure 11-9.** Creating a Java Class

A Java source file `CreateMongoDBDocument.java` gets added to the MongoDB project as shown in Figure 11-10.



**Figure 11-10.** Java Class in Java Project

As a summary, to add a document to MongoDB server a connection to MongoDB is first established. Subsequently, a Java class representation of a MongoDB database instance is created, and a Java class representation of a MongoDB collection is created. A MongoDB document Java class representation is created, and the document is added to the collection.

11. A connection to MongoDB is represented with the `com.mongodb.MongoClient` class. Follow these steps:
  - a. Create an instance of `MongoClient` using the `MongoClient(List<ServerAddress> seeds)` constructor.
  - b. Create a `List<ServerAddress>` using the host name as `10.0.2.15` and port as `27017`.
  - c. A logical database on the MongoDB server is represented with the `com.mongodb.DB` class. Create a `DB` instance using the `getDB(String dbname)` method in `MongoClient` with database name as `test`.
  - d. The `DBCollection` class represents a collection of document objects in a database. Create a `DBCollection` instance using the `MongoClient` method `createCollection(String name, DBObject options)`.
  - e. A key-value map for a BSON document object in a database collection is represented with the `DBObject` interface. The `BasicDBObject` class implements the `DBObject` interface and provides constructors to create a document object for a key/value. Create a document object using the `BasicDBObject(String key, Object value)` constructor and add key/value pairs to the object using the `append(String key, Object val)` method.
  - f. An instance of `BasicDBObject` represents a document object in a database collection. Create three `BasicDBObject` instances from the log data to be added to the MongoDB database collection.
  - g. The `DBCollection` class provides overloaded insert methods for adding a `BasicDBObject` instance to a collection. Add `BasicDBObject` instances to a `DBCollection` using the `insert(DBObject... arr)` method.

- h. To find a document object in the collection, invoke the `findOne()` method on the `DBCollection` object.

The `CreateMongoDBDocument` class is listed:

```

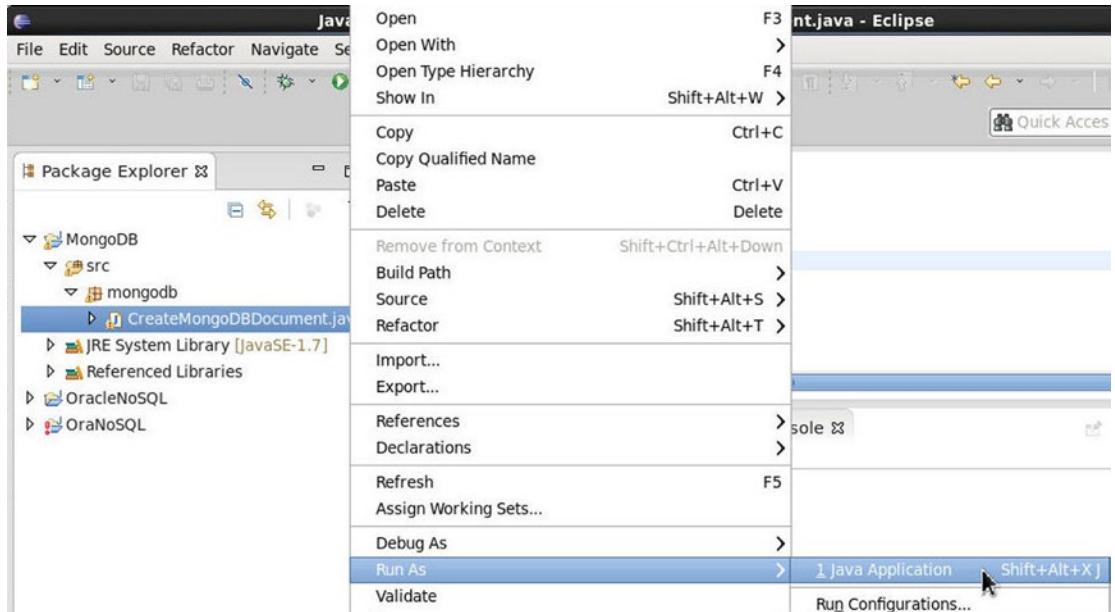
package mongodb;
import com.mongodb.MongoClient;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.ServerAddress;
import java.util.Arrays;
import java.net.UnknownHostException;

public class CreateMongoDBDocument {
 public static void main(String[] args) {
 try {
 MongoClient mongoClient = new MongoClient(
 Arrays.asList(new ServerAddress("10.0.2.15", 27017)));
 DB db = mongoClient.getDB("test");
 DBCollection coll = db.createCollection("wlslog", null);
 BasicDBObject row1 = new BasicDBObject("TIME_STAMP",
 "Apr-8-2014-7:06:16-PM-PDT").append("CATEGORY", "Notice")
 .append("TYPE", "WebLogicServer")
 .append("SERVERNAME", "AdminServer")
 .append("CODE", "BEA-000365").append("MSG", "Server state
changed to STANDBY");
 coll.insert(row1);
 BasicDBObject row2 = new BasicDBObject("TIME_STAMP",
 "Apr-8-2014-7:06:17-PM-PDT").append("CATEGORY", "Notice")
 .append("TYPE", "WebLogicServer")
 .append("SERVERNAME", "AdminServer")
 .append("CODE", "BEA-000365").append("MSG", "Server state
changed to STARTING");
 coll.insert(row2);
 BasicDBObject row3 = new BasicDBObject("TIME_STAMP",
 "Apr-8-2014-7:06:18-PM-PDT").append("CATEGORY", "Notice")
 .append("TYPE", "WebLogicServer")
 .append("SERVERNAME", "AdminServer")
 .append("CODE", "BEA-000360").append("MSG", "Server
started in RUNNING mode");
 coll.insert(row3);
 DBObject catalog = coll.findOne();
 System.out.println(row1);

 } catch (UnknownHostException e) {
 e.printStackTrace();
 }
 }
}

```

12. To add the BSON document objects to MongoDB run the Java application.  
 Right-click on `CreateMongoDBDocument.java` source file and select  
 Run As ► Java Application as shown in Figure 11-11.



**Figure 11-11.** Running the Java Application `CreateMongoDBDocument.java`

A MongoDB document store is created. One of the document objects added to the document store gets output in the Eclipse IDE Console as shown in Figure 11-12.



**Figure 11-12.** The MongoDB Object Added

Each BSON document object has an `_id` key added automatically.

```
{ "_id" : { "$oid" : "53a6cf25e4b09cac451ef1d6" } , "TIME_STAMP" : "Apr-8-2014-7:06:16-PM-PDT" , "CATEGORY" : "Notice" , "TYPE" : "WebLogicServer" , "SERVERNAME" : "AdminServer" , "CODE" : "BEA-000365" , "MSG" : "Server state changed to STANDBY" }
```

## Creating an External Table in Hive

Having created a MongoDB datastore we shall create a Hive external table defined on the MongoDB datastore using the Hive MongoDB Storage Handler. The TBLPROPERTIES for the MongoDB storage handler requires the mongo.user and mongo.password properties for which we need to create a user. We'll create a MongoDB user first and then go on to create the external table.

1. Start the MongoDB server shell with the following command.

```
>mongod
```

2. A user that creates another user must have the createUser action privilege. First, create an administrative user, which has the createUser action privilege. The use admin command sets the database as admin. Add a user called hive with the db.addUser() or db.createUser() method. Use the db.auth() method to authenticate the administrative user.

```
>use admin
>db.addUser('hive', 'hive');
>db.auth('hive','hive');
```

The output from the Mongo shell commands is shown in Figure 11-13.

```
> use admin
switched to db admin
> db.addUser('hive', 'hive');
WARNING: The 'addUser' shell helper is DEPRECATED. Please use 'createUser' instead
Successfully added user: { "user" : "hive", "roles" : ["root"] }
> db.auth('hive','hive');
1
> ■
```

**Figure 11-13.** Running Mongo Shell Commands

3. Shut down the server.

```
>db.shutdownServer()
```

4. Start the MongoDB server and log in to the shell as the administrative user hive with the following command.

```
mongo --port 27017 -u hive -p hive --authenticationDatabase admin
```

5. The MongoDB shell connects to the test database. Create a user called hive with the following command.

```
db.createUser({ "user" : "hive",
 "pwd": "hive",
 "roles" : [{ role: "clusterAdmin", db: "admin" },
```

```

 { role: "readAnyDatabase", db: "admin" },
 "readWrite"
] },
{ w: "majority" , wtimeout: 5000 })

```

A new user called `hive` gets added as shown in Figure 11-14.

The screenshot shows a terminal window with the title bar 'root@localhost:/mongodb'. The window contains the following text:

```

File Edit View Search Terminal Help
[root@localhost mongodb]# mongo --port 27017 -u hive -p hive --authenticationDatabase admin
MongoDB shell version: 2.6.3
connecting to: 127.0.0.1:27017/test
Server has startup warnings:
2014-06-22T10:49:07.407-0400 [initandlisten]
2014-06-22T10:49:07.411-0400 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary.
2014-06-22T10:49:07.412-0400 [initandlisten] ** 32 bit builds are limited
to less than 2GB of data (or less with --journal).
2014-06-22T10:49:07.412-0400 [initandlisten] ** Note that journaling defau
lts to off for 32 bit and is currently off.
2014-06-22T10:49:07.413-0400 [initandlisten] ** See http://dochub.mongodb.org/core/32bit
2014-06-22T10:49:07.413-0400 [initandlisten]
> db.createUser({ "user" : "hive",
... "pwd": "hive",
... "roles" : [{ role: "clusterAdmin", db: "admin" },
... { role: "readAnyDatabase", db: "admin" },
... "readWrite"
...],
... { w: "majority" , wtimeout: 5000 })
Successfully added user: {
 "user" : "hive",
 "roles" : [
 {
 "role" : "clusterAdmin",
 "db" : "admin"
 },
 {
 "role" : "readAnyDatabase",

```

**Figure 11-14.** Adding a New User

- As we shall be using the Remote metastore we need to start the Hive server with the following command.

```
hive -service hiveserver
```

The Hive Thrift Server gets started as shown in Figure 11-15.

```
[root@localhost mongodb]# hive --service hiveserver
Starting Hive Thrift Server
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/mongodb/hadoop-2.0.0-cdh4.6.0/share/hadoop/co
mmon/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/mongodb/hive-0.10.0-cdh4.6.0/lib/slf4j-log4j1
2-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
```

**Figure 11-15.** Starting the Hive Thrift Server

7. Start the Hive shell with the following command.

```
hive
```

8. In the Hive shell add the MongoDB storage handler for Hive to the Hive classpath with the ADD JAR command.

```
hive> ADD JAR hive-mongo-0.0.3-jar-with-dependencies.jar;
```

9. Run a CREATE EXTERNAL TABLE command to create the Hive table wlslog with the following included:

- Set the columns to TIME\_STAMP,CATEGORY,TYPE,SERVERNAME,CODE, and MSG.
- Set the STORED BY clause to 'org.yong3.hive.mongo.MongoStorageHandler'.
- In the WITH SERDEPROPERTIES clause set the mongo.column.mapping property to the column names in the MongoDB document collection.
- In the TBLPROPERTIES clause set the properties shown in Table 11-2.

**Table 11-2.** Hive Table TBLPROPERTIES Properties

Property	Value
mongo.host	10.0.2.15
mongo.port	27017
mongo.db	test
mongo.user	hive
mongo.passwd	hive
mongo.collection	wlslog

The command in the Hive shell to create a Hive external table is `wlslog`.

```
hive>CREATE EXTERNAL TABLE wlslog (TIME_STAMP string, CATEGORY string, TYPE
string, SERVERNAME string, CODE string, MSG string)
STORED BY 'org.yong3.hive.mongo.MongoStorageHandler'
WITH SERDEPROPERTIES ("mongo.column.mapping"="TIME_STAMP,CATEGORY,TYPE,SERV
ERNAME,CODE,MSG")
TBLPROPERTIES ("mongo.host" = "10.0.2.15", "mongo.port" = "27017",
"mongo.db" = "test", "mongo.user" = "hive", "mongo.passwd" = "hive",
"mongo.collection" = "wlslog");
```

The output from the command indicates that a Hive external table gets created.

- Run a SELECT query on the `wlslog` table to list the MongoDB data as shown in Figure 11-16.

```
hive> ADD JAR hive-mongo-0.0.3-jar-with-dependencies.jar;
Added hive-mongo-0.0.3-jar-with-dependencies.jar to class path
Added resource: hive-mongo-0.0.3-jar-with-dependencies.jar
hive> CREATE EXTERNAL TABLE wlslog (TIME_STAMP string, CATEGORY string, TYPE str
ing, SERVERNAME string, CODE string, MSG string)
> STORED BY 'org.yong3.hive.mongo.MongoStorageHandler'
> WITH SERDEPROPERTIES ("mongo.column.mapping"="TIME_STAMP,CATEGORY,TY
PE,SERVERNAME,CODE,MSG")
> TBLPROPERTIES ("mongo.host" = "10.0.2.15", "mongo.port" = "27017",
> "mongo.db" = "test", "mongo.user" = "hive", "mongo.passwd" = "hive", "mong
o.collection" = "wlslog");
OK
Time taken: 3.565 seconds
hive> SELECT * FROM wlslog;
OK
Apr-8-2014-7:06:16-PM-PDT Notice WebLogicServer AdminServer BEA-0003
65 Server state changed to STANDBY
Apr-8-2014-7:06:17-PM-PDT Notice WebLogicServer AdminServer BEA-0003
65 Server state changed to STARTING
Apr-8-2014-7:06:18-PM-PDT Notice WebLogicServer AdminServer BEA-0003
60 Server started in RUNNING mode
Time taken: 1.923 seconds
hive> ■
```

**Figure 11-16.** Creating Hive External Table

The `show tables` command should list the `wlslog` table added as shown in Figure 11-17.

```
hive> show tables;
OK
wlslog
Time taken: 22.479 seconds
hive> ■
```

**Figure 11-17.** Listing Hive Tables

## Summary

In this chapter we created a Hive external table on MongoDB using the MongoDB Storage Handler for Hive. A Hive external table is suitable if the data is to be managed externally, as by MongoDB database in this chapter. In the next chapter we shall integrate MongoDB data into Oracle Database in Oracle Data Integrator.

## CHAPTER 12



# Integrating MongoDB with Oracle Database in Oracle Data Integrator

MongoDB is a BSON (binary JSON) format NoSQL database and the leading NoSQL database. Oracle Database is the leading relational database. The use case could be to load MongoDB data into Oracle Database. Oracle Loader for Hadoop does not support the MongoDB BSON format directly. Oracle Loader for Hadoop supports loading from Hive. A Hive external table may be defined on a MongoDB datastore using the Hive Storage Handler for MongoDB. Oracle Data Integrator automates the use of Oracle Loader for Hadoop with the IKM File-Hive to Oracle knowledge module. In this chapter we shall load MongoDB data into Oracle Database in Oracle Data Integrator. This chapter is a continuation of Chapter 11. This chapter covers the following topics:

- Setting up the environment
- Creating the physical architecture
- Creating the logical architecture
- Creating the data models
- Creating the integration project
- Creating the integration interface
- Running the interface
- Selecting integrated data in an Oracle Database table

## Setting Up the Environment

In addition to the software used in Chapter 11, download and install the following software.

- Oracle Database 11g. This chapter assumes you have Oracle Database 11g (or later) installed.
- Oracle Loader for Hadoop 3.0.0. Download from [www.oracle.com/technetwork/database/database-technologies/bdc/big-data-connectors/downloads/index.html](http://www.oracle.com/technetwork/database/database-technologies/bdc/big-data-connectors/downloads/index.html) as part of the “Big Data Collectors” download.
- Oracle Data Integrator 11g. Download Data Integrator 11g (or later) from [www.oracle.com/technetwork/middleware/data-integrator/downloads/index.html](http://www.oracle.com/technetwork/middleware/data-integrator/downloads/index.html).

Oracle Linux is used in this chapter as in Chapter 11. The installation procedure for Oracle Database 11g and Oracle Data Integrator 11g is too elaborate to be included in a chapter. Download Oracle Loader for Hadoop Release 3.0.0 oraloader-3.0.0.x86\_64.zip from [www.oracle.com/technetwork/database/database-technologies/bdc/big-data-connectors/downloads/index.html](http://www.oracle.com/technetwork/database/database-technologies/bdc/big-data-connectors/downloads/index.html). Unzip the file to a directory. Two files get extracted oraloader-3.0.0-h1.x86\_64.zip and oraloader-3.0.0-h2.x86\_64.zip. The oraloader-3.0.0-h1.x86\_64.zip file is for Apache Hadoop 1.x and oraloader-3.0.0-h2.x86\_64.zip for CDH4 and CDH5. As we are using CDH4.6, extract oraloader-3.0.0-h2.x86\_64.zip on Linux as user root with the following command.

```
root>unzip oraloader-3.0.0-h2.x86_64.zip
```

Oracle Loader for Hadoop 3.0.0 gets extracted to oraloader-3.0.0-h2 directory. Set the environment variables for Oracle Database, Oracle Data Integrator, Oracle Loader for Hadoop, Hadoop, Hive, Java, and MongoDB in the bash shell.

```
vi ~/.bashrc
```

```
export ODI_HOME=/home/dvohra/dbhome_1
export ORACLE_HOME=/home/oracle/app/oracle/product/11.2.0/dbhome_1
export ORACLE_SID=ORCL
export OLH_HOME=/mongodb/oraloader-3.0.0-h2
export HADOOP_PREFIX=/mongodb/hadoop-2.0.0-cdh4.6.0
export HADOOP_CONF=$HADOOP_PREFIX/etc/hadoop
export MONGO_HOME=/mongodb/mongodb-linux-i686-2.6.3
export HIVE_HOME=/mongodb/hive-0.10.0-cdh4.6.0
export HIVE_CONF=$HIVE_HOME/conf
export JAVA_HOME=/mongodb/jdk1.7.0_55
export HADOOP_MAPRED_HOME=/mongodb/hadoop-2.0.0-cdh4.6.0/bin
export HADOOP_HOME=/mongodb/hadoop-2.0.0-cdh4.6.0/share/hadoop/mapreduce
export HADOOP_CLASSPATH=$HADOOP_HOME/*:$HADOOP_HOME/lib/*:$HIVE_HOME/lib/*:$OLH_HOME/jlib/*:
/mongodb/mongo-java-driver-2.6.3.jar:/mongodb/hive-mongo-0.0.3-jar-with-dependencies.
jar:$HIVE_CONF:$HADOOP_CONF
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_MAPRED_HOME::$HIVE_HOME/bin:$MONGO_HOME/
bin:$ORACLE_HOME/bin
```

We shall use the Hive table created over MongoDB in Chapter 11 to integrate MongoDB data to Oracle Database. Create the target database table in Oracle Database with the following SQL script.

```
CREATE TABLE OE.wlslog (time_stamp VARCHAR2(255), category VARCHAR2(255), type
VARCHAR2(255), servername VARCHAR2(255), code VARCHAR2(255), msg VARCHAR2(255));
```

The output from the CREATE TABLE SQL statement is shown in Figure 12-1. Oracle Database table OE.WLSLOG has the same column names as the Hive external table from which data is to be loaded.

```
SQL> CREATE TABLE OE.wlslog (time_stamp VARCHAR2(255), category VARCHAR2(255), type VARCHAR2(255), servername VARCHAR2(255), code VARCHAR2(255), msg VARCHAR2(255));
Table created.

SQL> DESC OE.WLSLOG;
Name Null? Type

TIME_STAMP VARCHAR2(255)
CATEGORY VARCHAR2(255)
TYPE VARCHAR2(255)
SERVERNAME VARCHAR2(255)
CODE VARCHAR2(255)
MSG VARCHAR2(255)

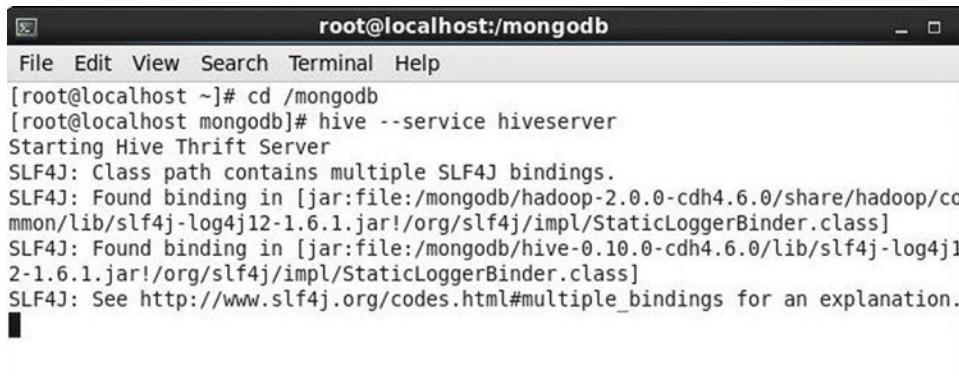
SQL>
```

**Figure 12-1.** Creating Oracle Database Table

As discussed in Chapter 11, add data to MongoDB datastore in a Java application. Create a Hive external table wlslog also as discussed in Chapter 11. Start the Hive remote metastore with the following command.

```
hive -service hiveserver
```

Hive server gets started as shown in Figure 12-2.



```
File Edit View Search Terminal Help
[root@localhost ~]# cd /mongodb
[root@localhost mongodb]# hive --service hiveserver
Starting Hive Thrift Server
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/mongodb/hadoop-2.0.0-cdh4.6.0/share/hadoop/common/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/mongodb/hive-0.10.0-cdh4.6.0/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
```

**Figure 12-2.** Starting Hive Server

As we are using Oracle Data Integrator (ODI), we need to install ODI, create the repositories, and connect to repositories for defining the physical and logical architecture and the models and the interface for mapping the source datastore to the target datastore. Oracle Data Integrator Studio is launched with the following command.

```
cd /home/dvohra/dbhome_1/oracledi/client
sh odi.sh
```

## Creating the Physical Architecture

The physical architecture consists of the data servers and the physical schemas. In this section we shall add data servers and physical schemas for MongoDB and Oracle Database. The physical and logical architecture for MongoDB is based on the Hive technology.

1. Select Topology ► Physical Architecture ► Technologies ► Hive in ODI Studio as shown in Figure 12-3.

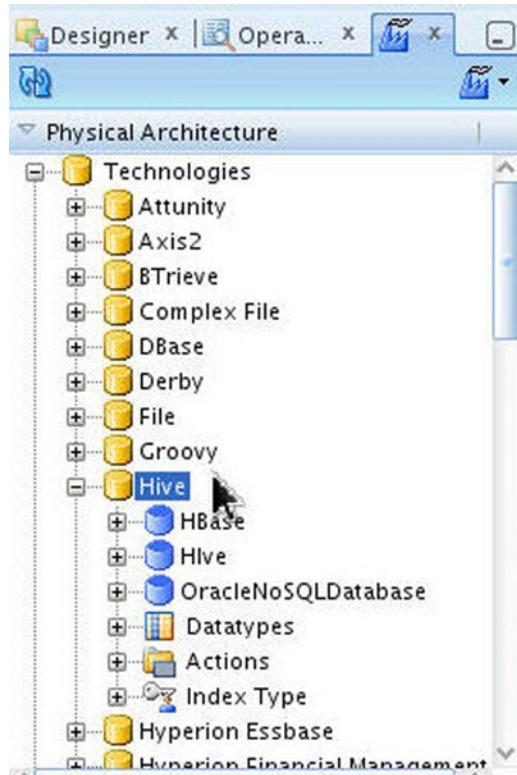


Figure 12-3. Selecting the Hive Technology in Physical Architecture

2. To create a new data server for MongoDB via the Hive table right-click on Hive and select New Data Server as shown in Figure 12-4.

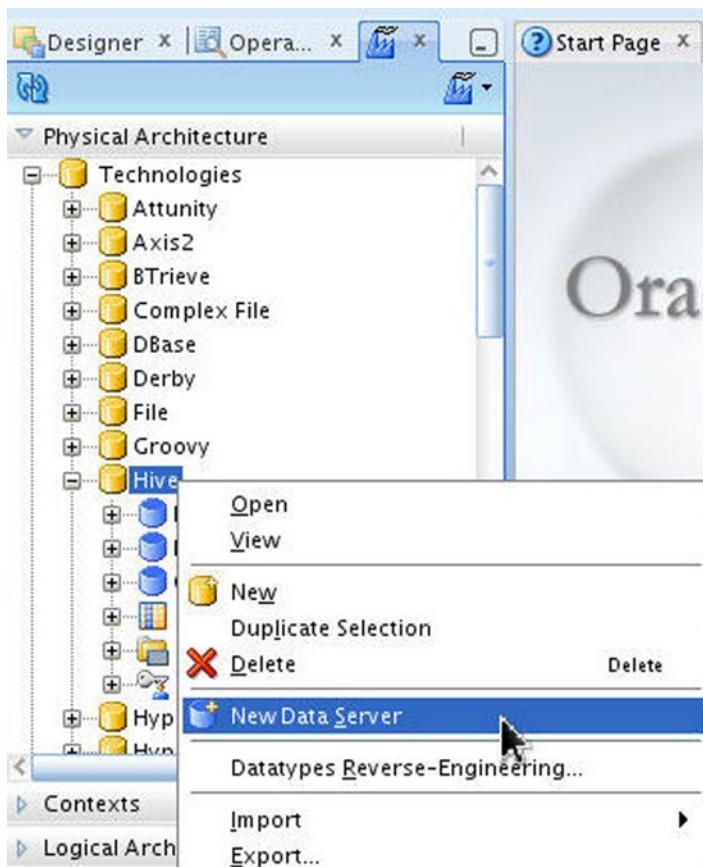
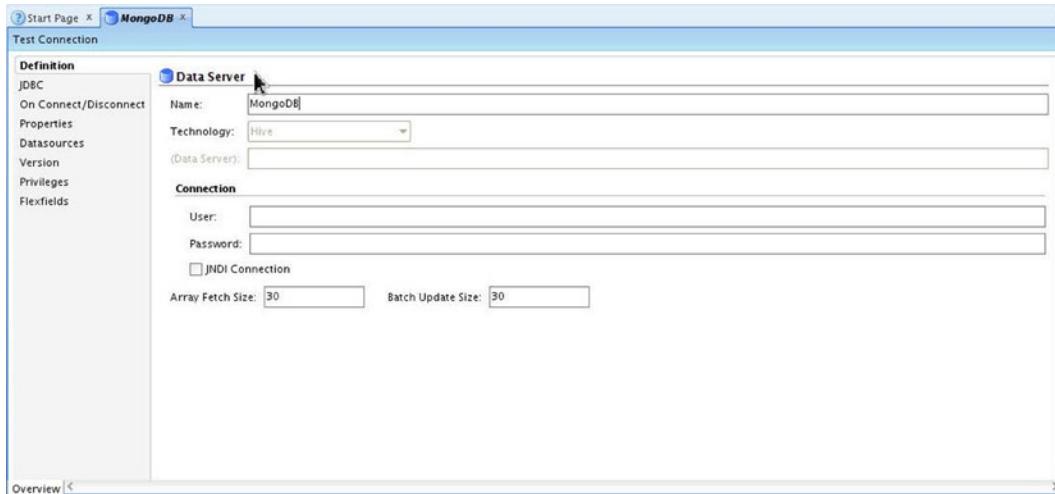


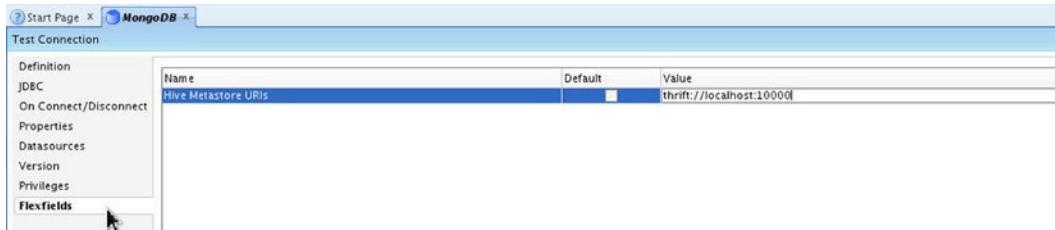
Figure 12-4. Creating a New Hive Data Server

3. In Data Server Definition specify a Name (MongoDB) as shown in Figure 12-5.  
The Technology is preselected as Hive.



**Figure 12-5.** Configuring New Data Server Definition

4. Select the Flexfields tab as shown in Figure 12-6. Deselect the Default checkbox and specify the Value for the Hive Metastore URIs as `thrift://localhost:10000`.



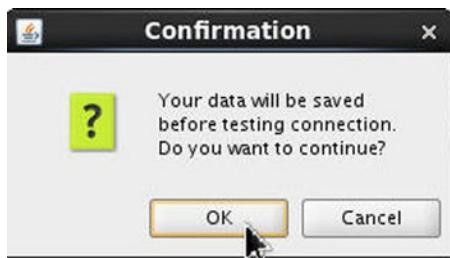
**Figure 12-6.** Configuring Hive Metastore URIs

5. Select the JDBC tab and select the JDBC Driver as the `HiveDriver` and specify the JDBC Url as `jdbc:hive://localhost:10000/default`. Click on Test Connection to test the JDBC connection as shown in Figure 12-7.



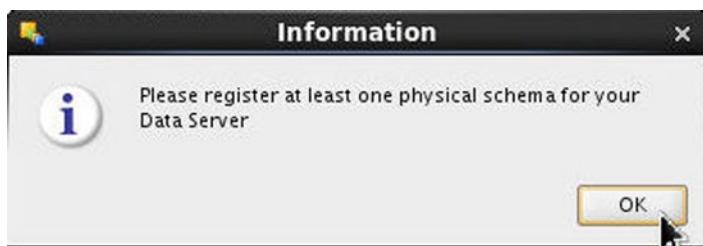
**Figure 12-7.** Testing Connection with Hive Server

- A Confirmation dialog indicates “Your data will be saved before testing connection. Do you want to continue?” as shown in Figure 12-8. Click on OK.



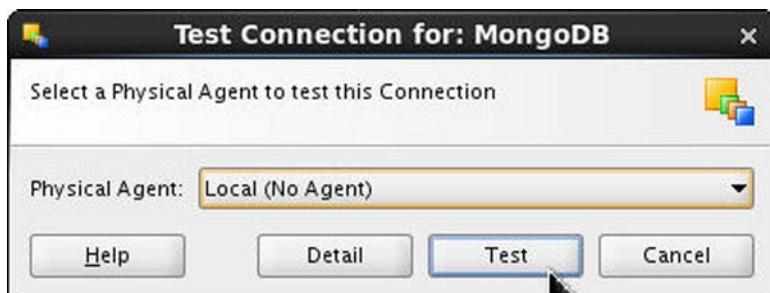
**Figure 12-8.** Confirmation Dialog for Testing Connection

- An Information dialog prompts to register at least one physical schema for the Data Server as shown in Figure 12-9. Click on OK.



**Figure 12-9.** Information Dialog to Register a Physical Schema

- In Test Connection click on Test as shown in Figure 12-10.



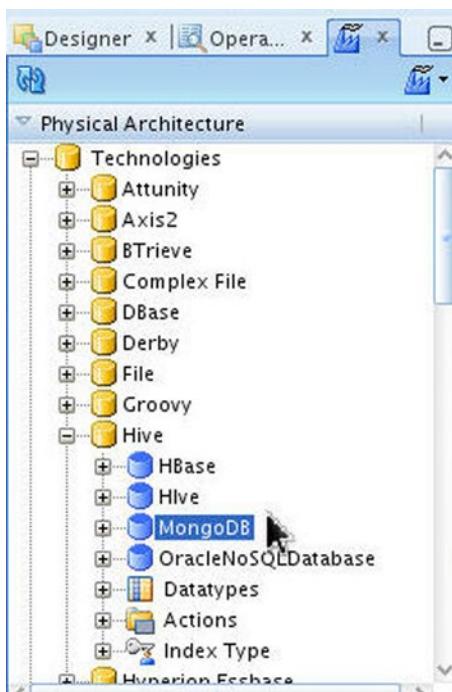
**Figure 12-10.** Testing Connection with Hive Server

9. A “Successful connection” message indicates that a connection gets established as shown in Figure 12-11. Click on OK.



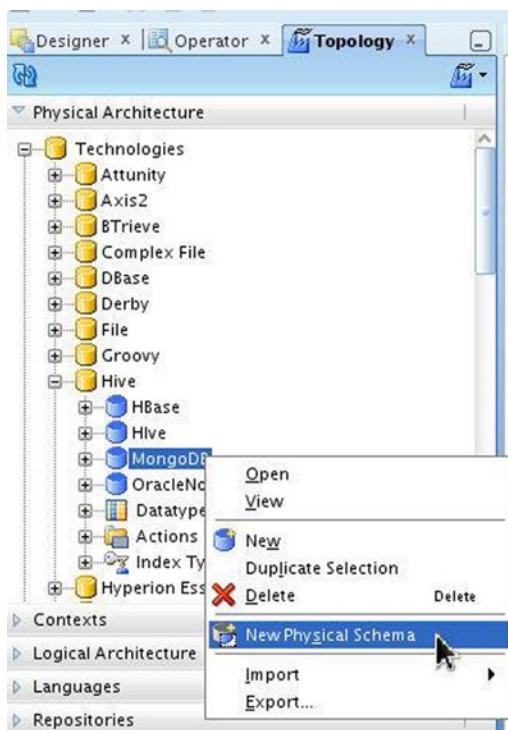
**Figure 12-11.** Information Dialog for a Successful Connection

10. Click on Save. A data server for MongoDB based on the Hive technology gets added as shown in Figure 12-12.



**Figure 12-12.** Hive Data Server Called MongoDB

11. Next, add a Physical Schema to the data server. Right-click on the MongoDB data server node and select New Physical Schema as shown in Figure 12-13.



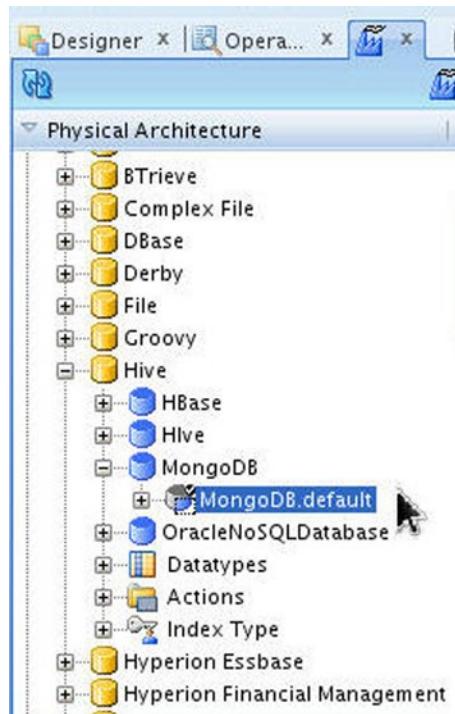
**Figure 12-13.** Selecting Physical Architecture > Hive > MongoDB > New Physical Schema

12. In the Physical Schema Definition the Name is prespecified as MongoDB.default. Specify Schema (Schema) and Schema (Work Schema) as default as shown in Figure 12-14. Click on Save.

The screenshot shows the 'Physical Schema [Data Server: MongoDB]' dialog. The 'Definition' tab is selected. The 'Name' field is populated with 'MongoDB.default'. The 'Schema (Schema)' and 'Schema (Work Schema)' dropdowns are both set to 'default'. The 'Default' checkbox is checked. The 'Work Tables Prefix' section contains fields for Errors, Loading, Integration, and Temporary Indexes. The 'Journalizing elements prefixes' section has fields for Datastores, Views, and Triggers. The 'Naming Rules' section contains fields for Local Object Mask, Remote Object Mask, Partition Mask, Sub-Partition Mask, Local Sequence Mask, and Remote Sequence Mask. The 'Local Object Mask' is set to '%SCHEMA.%OBJECT'.

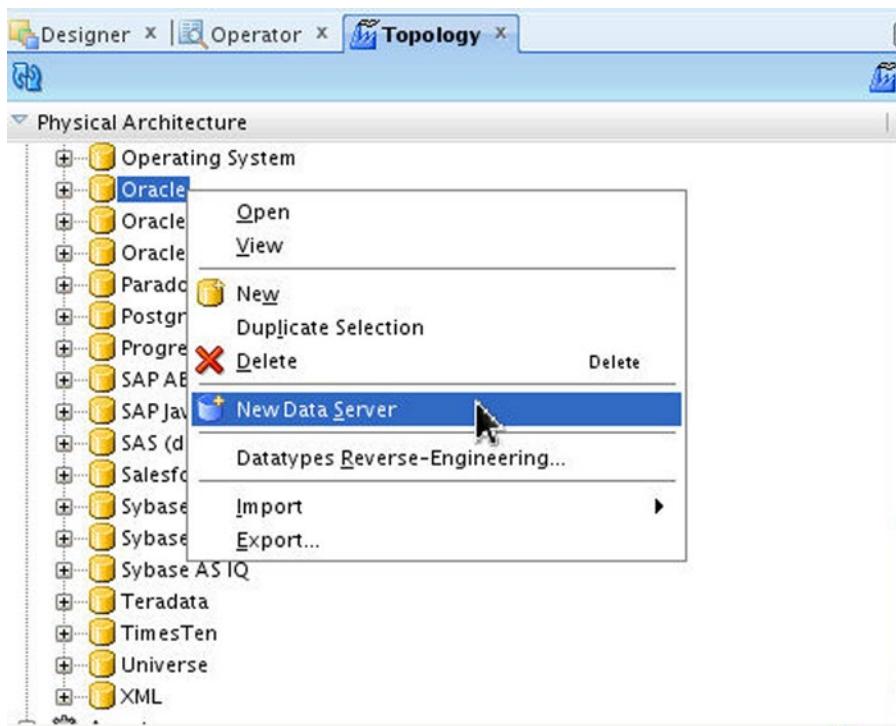
**Figure 12-14.** Physical Schema Definition for MongoDB Data Server

A Physical Schema gets added to the data server as shown in Figure 12-15.



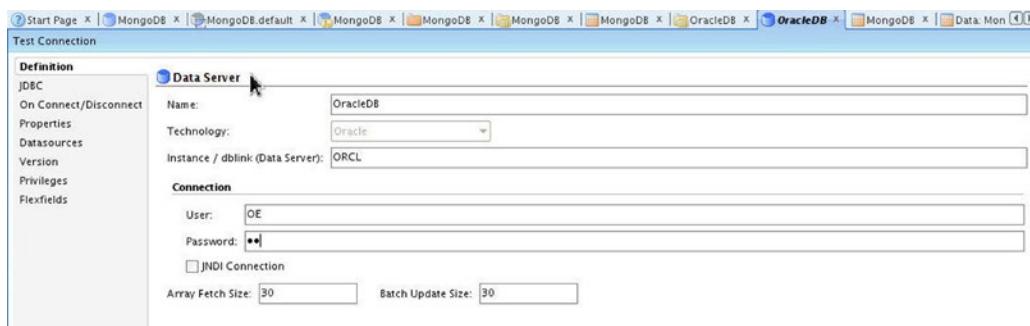
**Figure 12-15.** Physical Schema MongoDB.default

13. The target datastore is an Oracle Database table. To create a data server for Oracle Database, right-click on Topology > Physical Architecture > Technologies > Oracle and select New Data Server as shown in Figure 12-16.



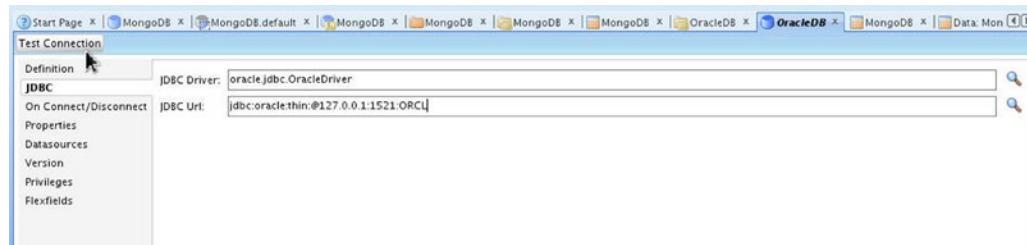
**Figure 12-16.** Selecting Physical Architecture ► Oracle ► New Data Server

14. In Data Server Definition specify a data server Name. The Technology is preselected as Oracle. Specify Instance as ORCL as shown in Figure 12-17. Specify User and Password.



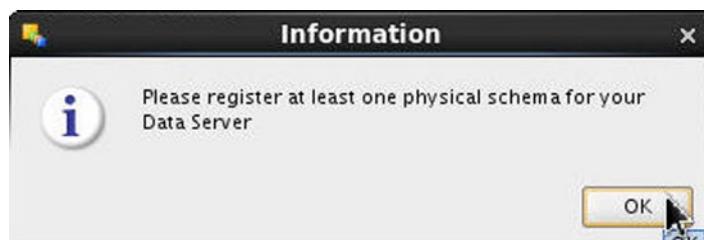
**Figure 12-17.** Configuring Data Server for Oracle Database

15. Select the JDBC tab. Select the JDBC Driver as `oracle.jdbc.OracleDriver` and specify JDBC Url as `jdbc:oracle:thin:@127.0.0.1:1521:ORCL`. Click on Test Connection to test the JDBC connection as shown in Figure 12-18.



**Figure 12-18.** Clicking on Test Connection

16. An Information dialog prompts to register at least one physical schema for the data server as shown in Figure 12-19. Click on OK.



**Figure 12-19.** Information Dialog to Register a Physical Schema

17. In Test Connection click on Test as shown in Figure 12-20.



**Figure 12-20.** Testing Connection with Oracle Database

18. A Successful Connection message indicates that a connection has been established as shown in Figure 12-21. Click on OK.



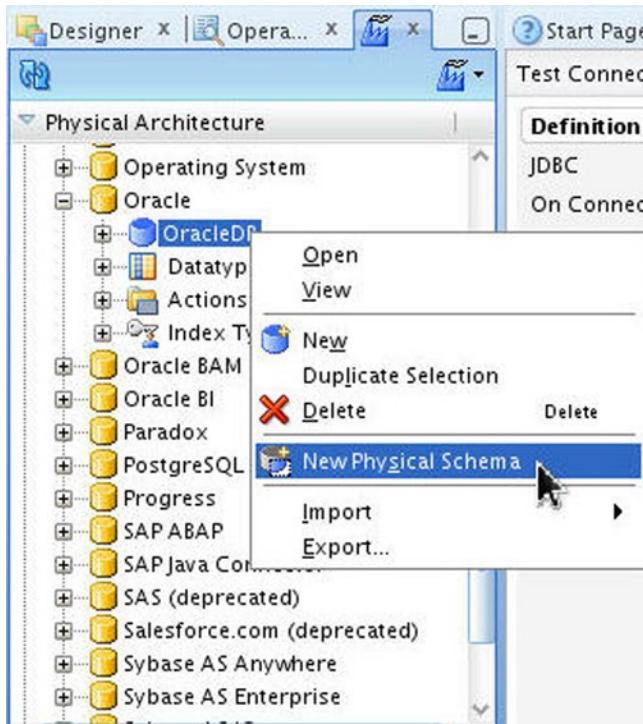
**Figure 12-21.** Information Dialog for a Successful Connection

An Oracle technology data server gets added as shown in Figure 12-22.



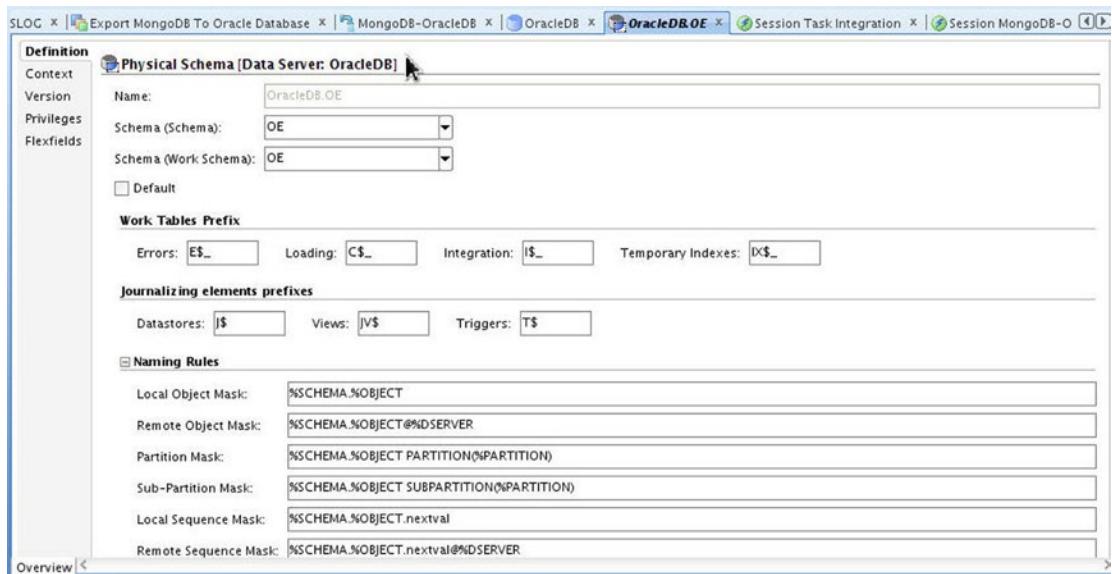
**Figure 12-22.** New Oracle Technology Data Server

19. To register a physical schema with the data server, right-click on the data server node and select New Physical Schema as shown in Figure 12-23.



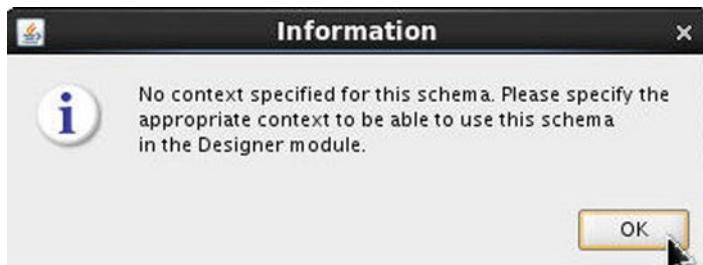
**Figure 12-23.** Selecting Physical Architecture ► Oracle ► OracleDB Data Server ► New Physical Schema

20. In Physical Schema Definition the Name is prespecified. Specify Schema (Schema) and Schema (Work Schema) as the User used to connect to Oracle Database, which is OE for the data server OracleDB configured earlier as shown in Figure 12-24. Click on Save.



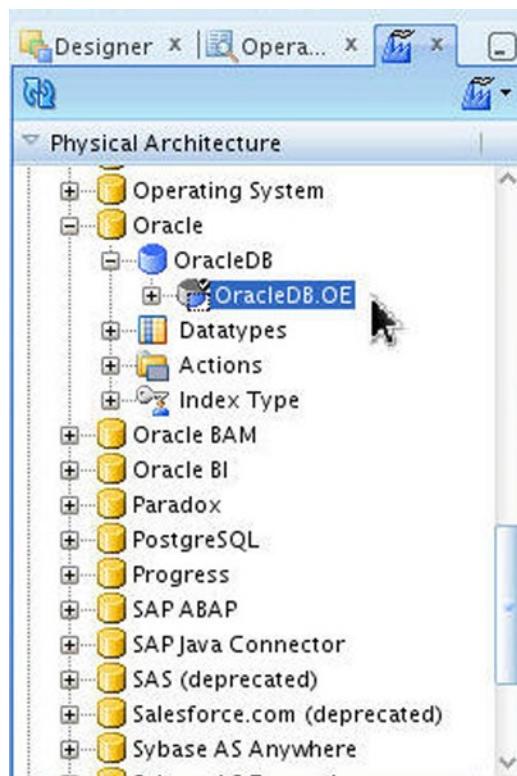
**Figure 12-24.** Configuring Physical Schema for OracleDB Data Server

21. An Information dialog prompts to specify a context for the schema as shown in Figure 12-25. Click on OK.



**Figure 12-25.** Information Dialog Prompting to Specify a Context

A physical schema gets added to the data server as shown in Figure 12-26.

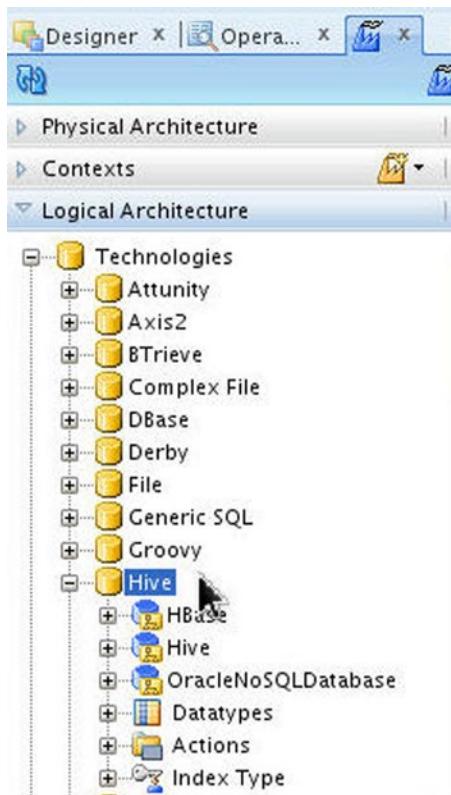


**Figure 12-26.** Physical Schema for Oracle Database

## Creating the Logical Architecture

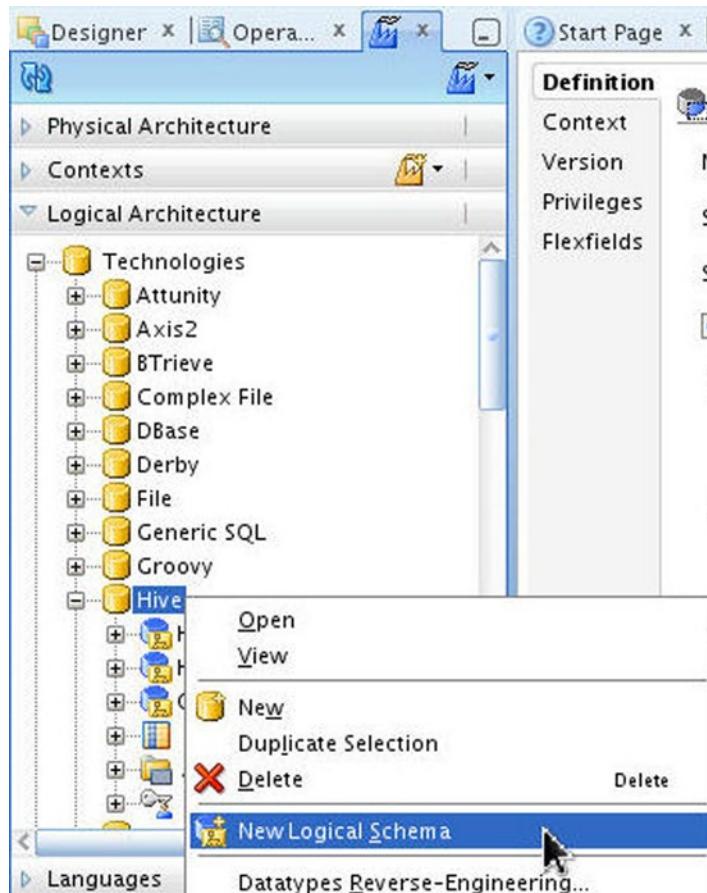
A logical schema represents the ODI's interface to the physical schema. In this section we shall create logical schemas for the Oracle technology data server and the Hive technology data server.

1. Select Topology ► Logical Architecture ► Technologies ► Hive as shown in Figure 12-27.



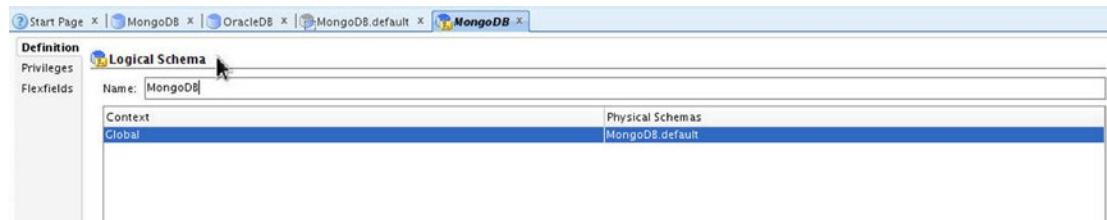
**Figure 12-27.** Selecting Logical Architecture ► Technologies ► Hive

2. Right-click on Hive and select New Logical Schema as shown in Figure 12-28.



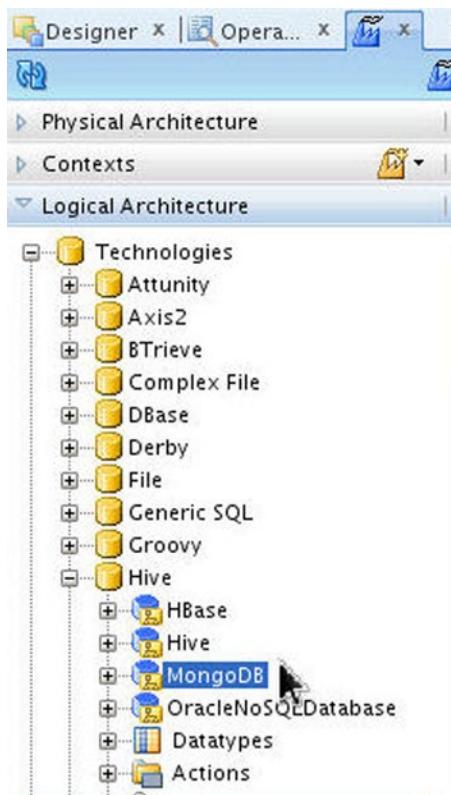
**Figure 12-28.** Selecting Hive ► New Logical Schema

3. In Logical Schema Definition specify a Name, MongoDB. In Context the Global context is listed. In Physical Schemas for the Global context select the physical schema for the Hive technology, MongoDB.default, configured earlier as shown in Figure 12-29. Click on Save.



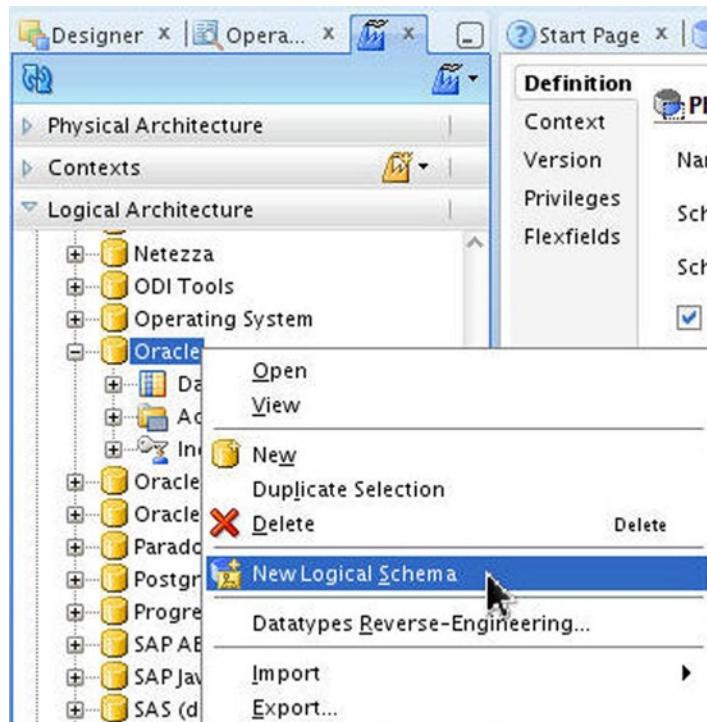
**Figure 12-29.** Configuring Logical Schema for Hive

A Hive technology logical schema gets added as shown in Figure 12-30.



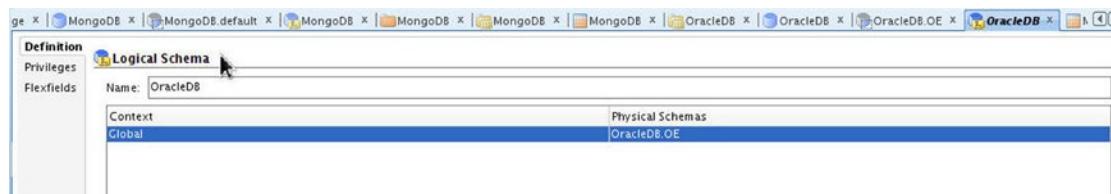
**Figure 12-30.** New Logical Schema for Hive Called MongoDB

- Similarly, go to Topology > Logical Architecture > Technologies > Oracle, right-click Oracle, and select New Logical Schema as shown in Figure 12-31.



**Figure 12-31.** Selecting Logical Architecture > Oracle > New Logical Schema

- In the Logical Schema Definition specify a Name, OracleDB. In Context the Global context is listed. Select the physical schema OracleDB.OE configured for the Oracle technology as shown in Figure 12-32.



**Figure 12-32.** Configuring Logical Schema for Oracle Database

A logical schema for Oracle technology gets added as shown in Figure 12-33.

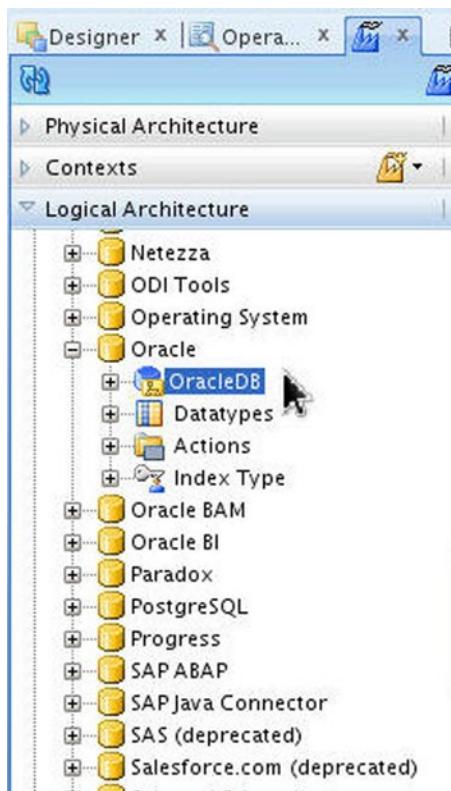
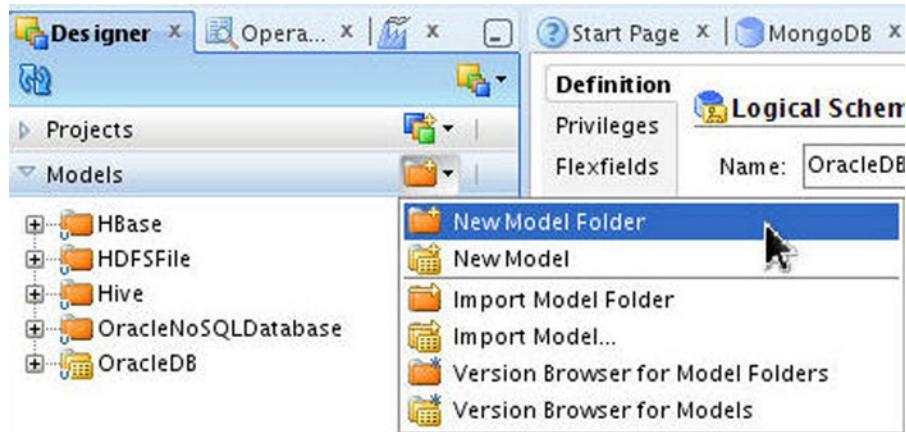


Figure 12-33. New Logical Schema for Oracle Technology

## Creating the Data Models

A Data model includes detailed information about data such as the columns in the data, the data types of each column, the logical length of data, whether the column is nullable, and so forth. In this section we shall create data models for MongoDB and Oracle Database, the source and target databases.

- To create a model for MongoDB datastore first create a model folder. Select Designer > Models and select New Model Folder as shown in Figure 12-34.



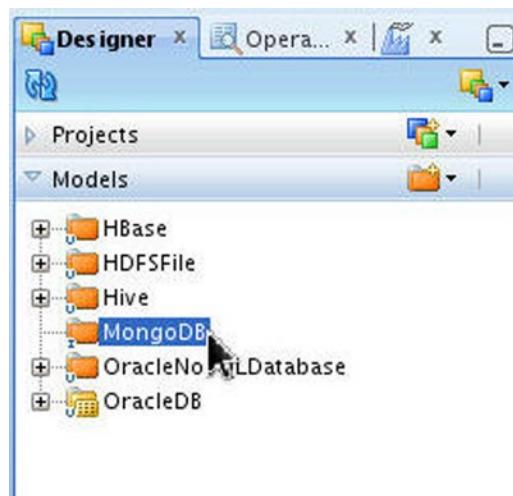
**Figure 12-34.** Selecting Models > New Model Folder

- In Model Folder Definition specify a Model Name as shown in Figure 12-35.



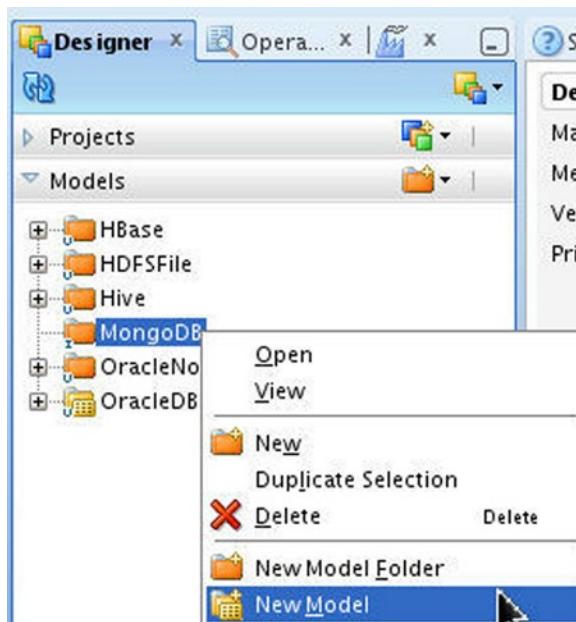
**Figure 12-35.** Configuring Model Folder Definition

A model folder gets added to the Models as shown in Figure 12-36.



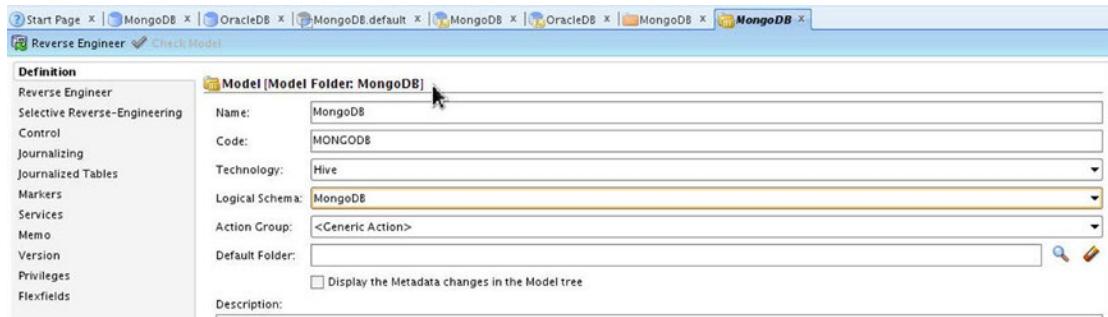
**Figure 12-36.** New Model Folder

3. Right-click on the model folder and select New Model as shown in Figure 12-37.



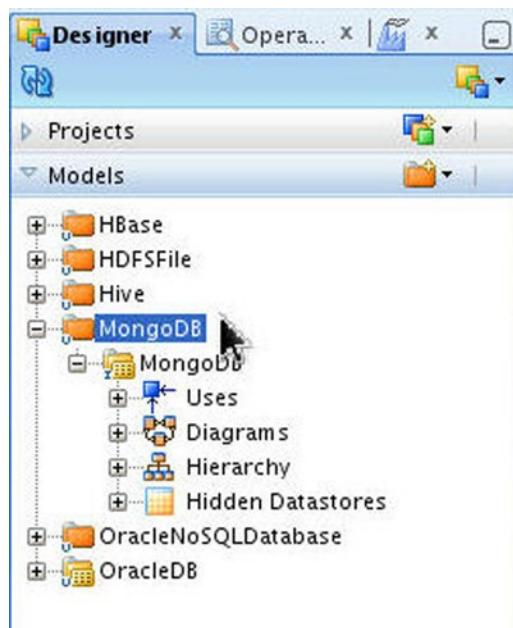
**Figure 12-37.** Selecting New Model

4. In the Model Definition specify a Name and select Technology as Hive as shown in Figure 12-38. Select Logical Schema as the MongoDB schema created in the previous section. Select Action Group as <Generic Action>. Click on Save.



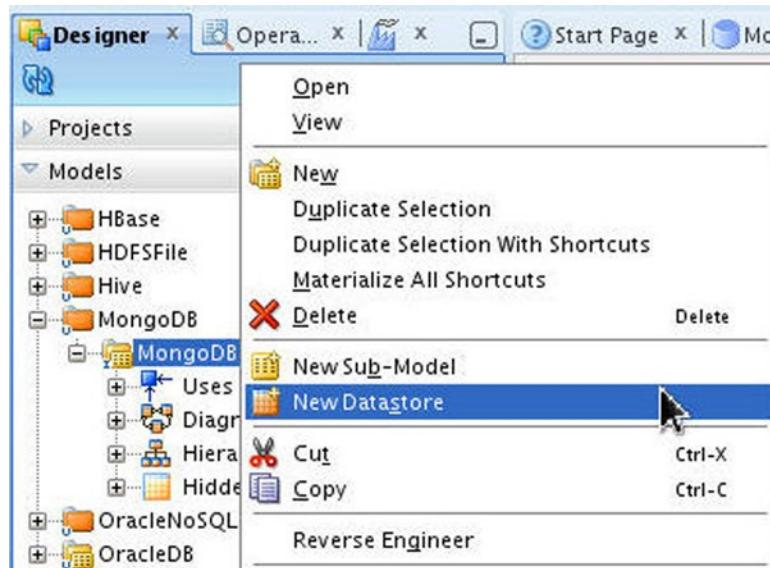
**Figure 12-38.** Configuring a Model for MongoDB

A model gets added for the Hive technology based on the logical schema MongoDB as shown in Figure 12-39.



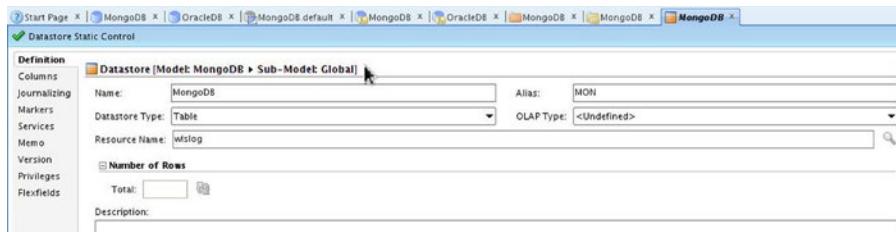
**Figure 12-39.** New Model Called MongoDB

5. Right-click on the model and select New Datastore as shown in Figure 12-40.



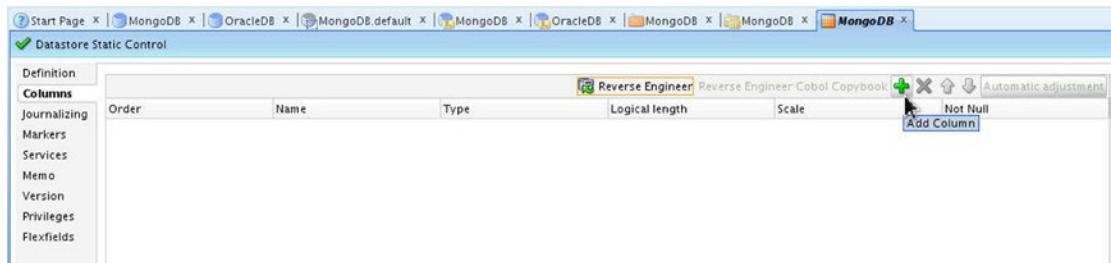
**Figure 12-40.** Selecting New Datastore

- In the Datastore definition specify a Name and select Datastore Type as Table. Specify Resource Name as wlslog as shown in Figure 12-41.



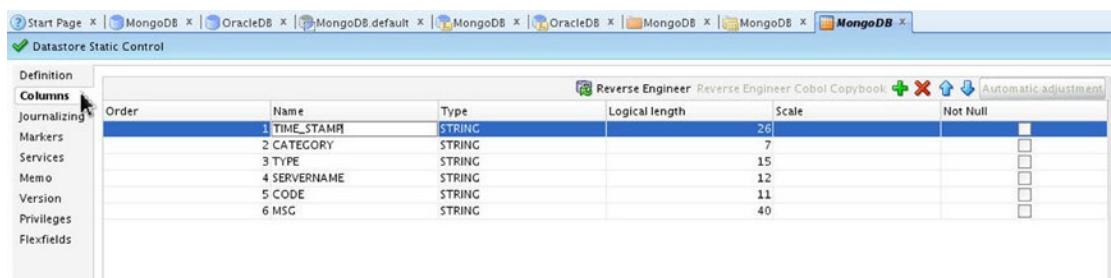
**Figure 12-41.** Configuring a New Datastore

- Select the Columns tab. We shall construct the data model for the datastore. Click on Add Column as shown in Figure 12-42.



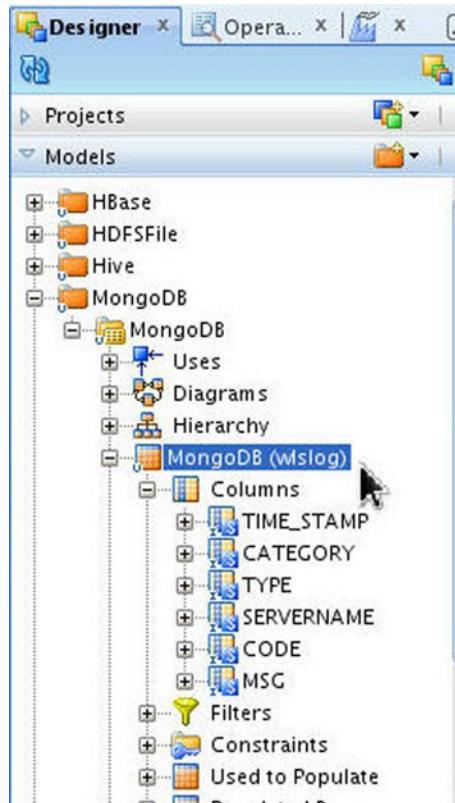
**Figure 12-42.** Selecting Add Column

- Specify the column Name, Type, Logical Length for the columns in the Hive table wlslog, which is stored as a MongoDB document collection as shown in Figure 12-43. Click on Save.



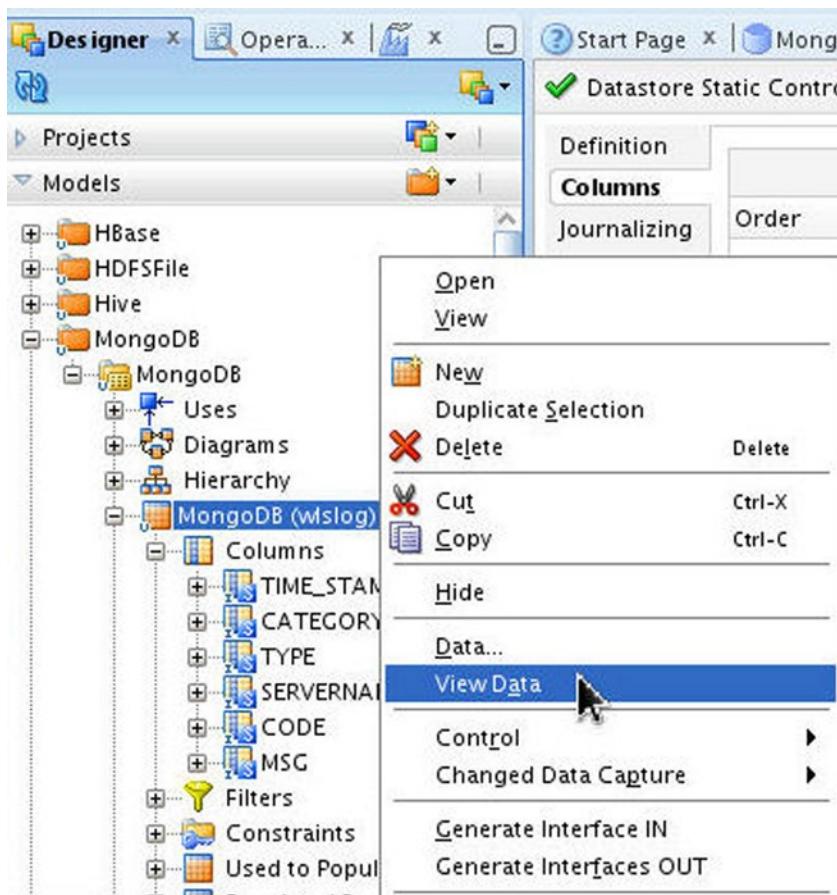
**Figure 12-43.** Configuring Columns

A datastore gets added to the model as shown in Figure 12-44.



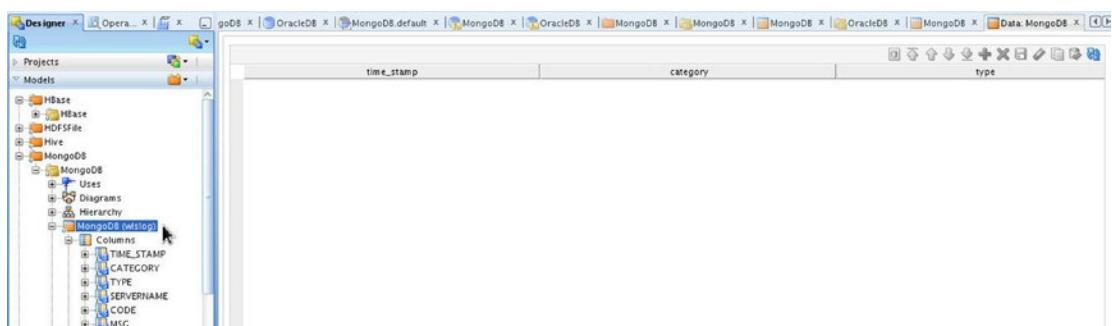
**Figure 12-44.** New Datastore

9. Initially the datastore is empty. Right-click on the datastore and select View Data as shown in Figure 12-45.



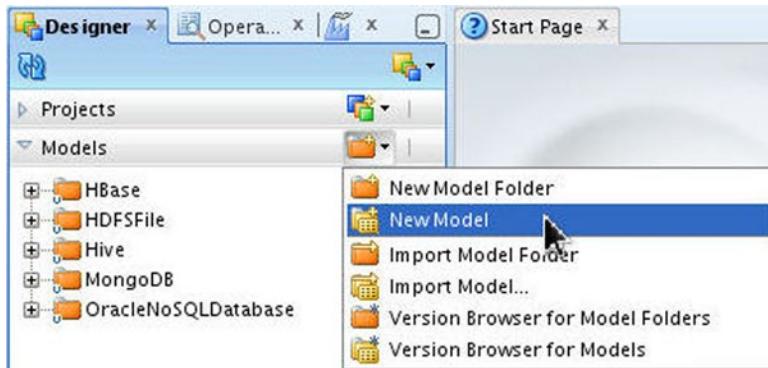
**Figure 12-45.** Selecting View Data

The empty datastore table gets displayed as shown in Figure 12-46.



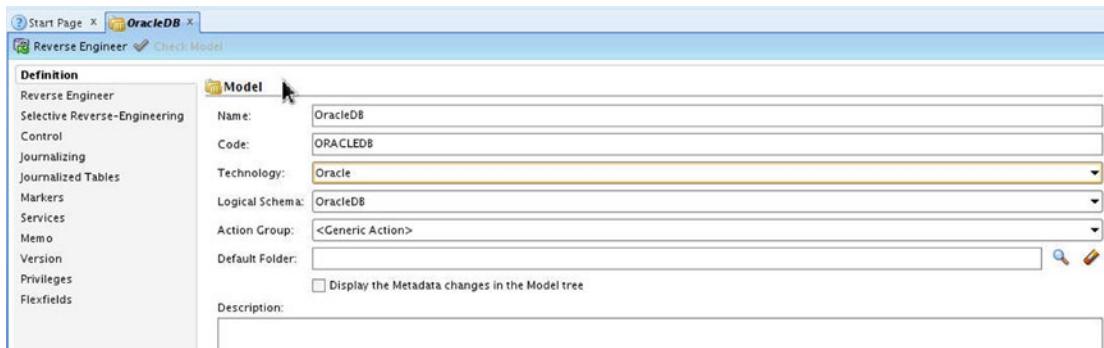
**Figure 12-46.** The Empty Datastore Table

10. Next, add a model for Oracle Database. Select New Model in Designer > Models as shown in Figure 12-47.



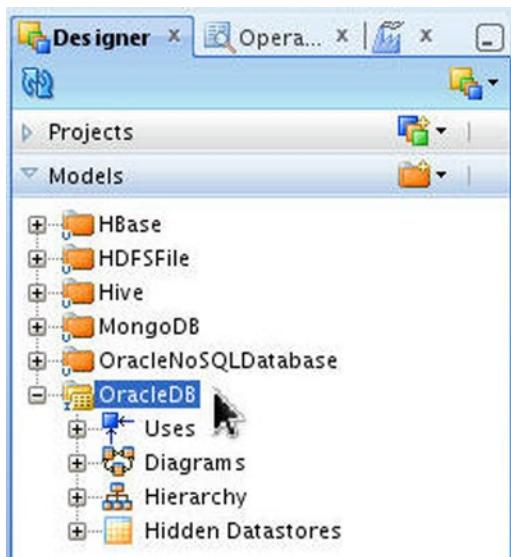
**Figure 12-47.** Adding New Model for Oracle Database

11. In Model Definition specify Name and select Technology as Oracle as shown in Figure 12-48. Select Logical Schema as the OracleDB schema created earlier. Select Action Group as <Generic Action>. Click on Save.



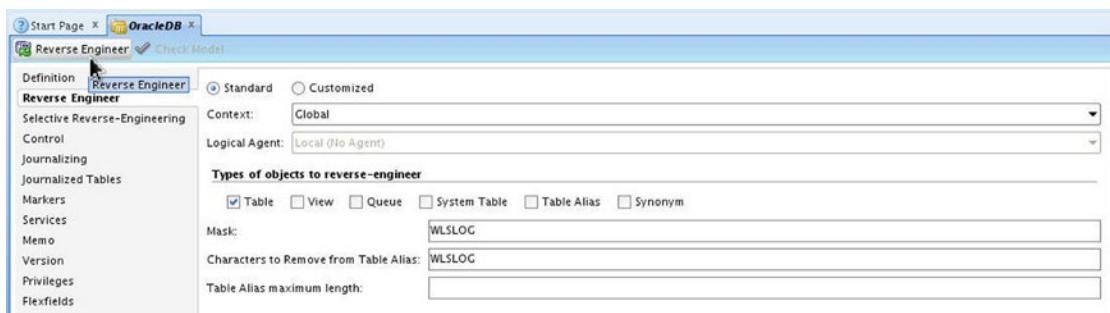
**Figure 12-48.** Configuring Model Definition for Oracle Database

A model for Oracle Database gets added as shown in Figure 12-49.



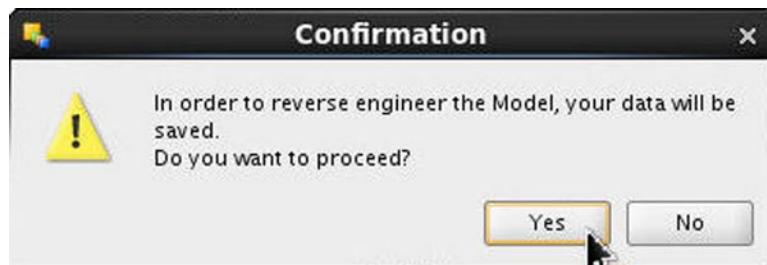
**Figure 12-49.** Model OracleDB

12. Next, we shall reverse engineer the datastore from Oracle Database.
  - a. Select the Reverse Engineer tab and select Standard.
  - b. Select Context as Global and select Types of objects to reverse engineer as Table.
  - c. Specify Mask as WLSLOG, which is the Oracle Database table to which MongoDB data is to be loaded.
  - d. Also specify Characters to remove from Table Alias as WLSLOG.
  - e. Click on Reverse Engineer as shown in Figure 12-50.



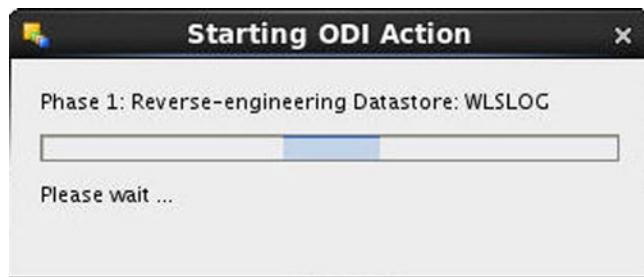
**Figure 12-50.** Selecting Reverse Engineer

13. Click on Yes in the Confirmation dialog as shown in Figure 12-51.



**Figure 12-51.** Confirmation Dialog for Reverse Engineering

The Reverse engineering of the WLSLOG table starts as shown in Figure 12-52.



**Figure 12-52.** Reverse Engineering

The WLSLOG database table gets reverse engineered as a WLSLOG datastore. The columns for the datastore are also reverse engineered as shown in Figure 12-53.

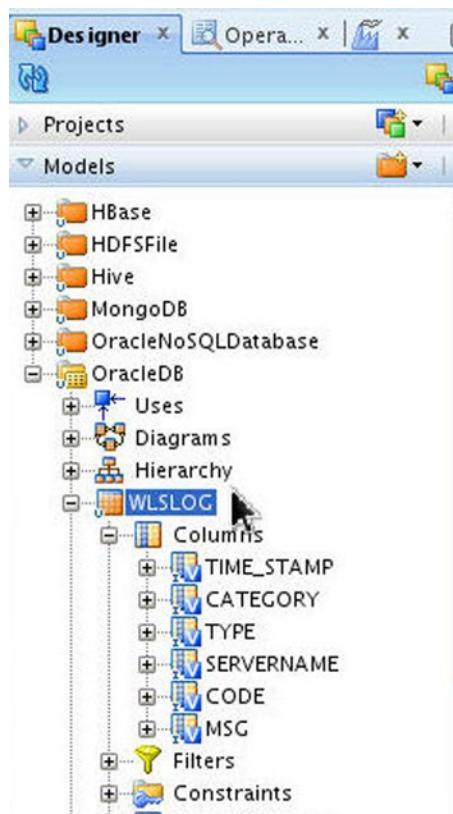
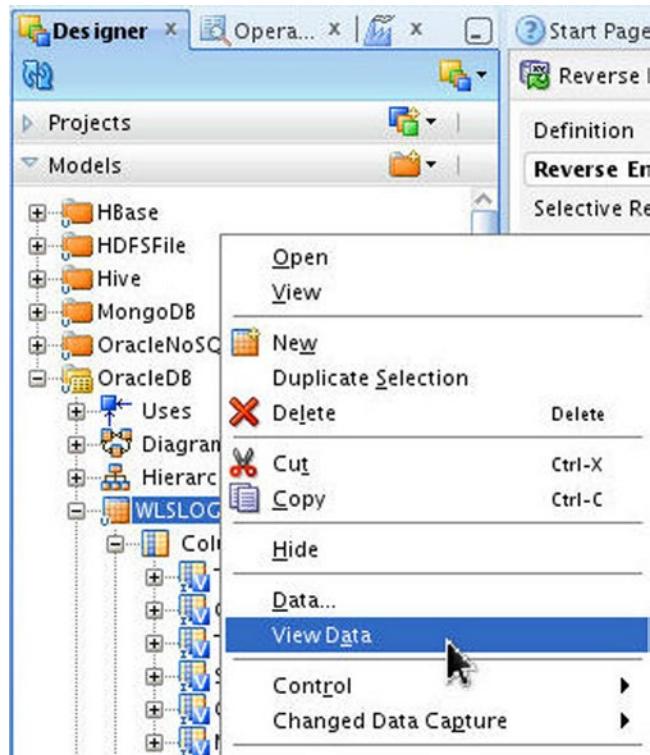


Figure 12-53. Reverse-Engineered Columns

14. Initially the datastore is empty. Right-click on the WLSLOG datastore and select View Data as shown in Figure 12-54.



**Figure 12-54.** Selecting View Data for Oracle Database Model

An empty table gets displayed for the datastore with just the column headers as shown in Figure 12-55.

TIME_STAMP	CATEGORY	TYPE

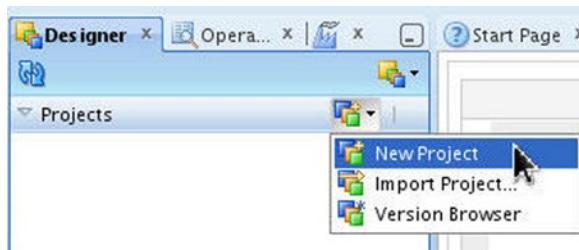
The screenshot shows the Oracle Data Integrator Designer interface with the 'Data: WLSLOG' tab active. The left sidebar shows the same model structure as Figure 12-54. The main pane displays a table with three columns: TIME\_STAMP, CATEGORY, and TYPE. The table is currently empty, with no data rows present.

**Figure 12-55.** Empty Data Table

## Creating the Integration Project

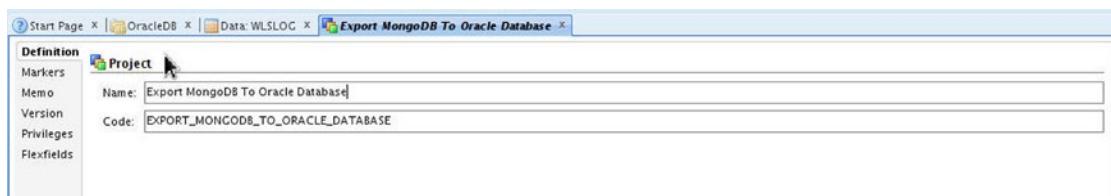
An integration project is required for integrating data from a source datastore to a target datastore. Next we shall create an integration project, add an integration interface, and run the integration interface.

1. Select Designer ► Projects and select New Project to create a new project as shown in Figure 12-56.



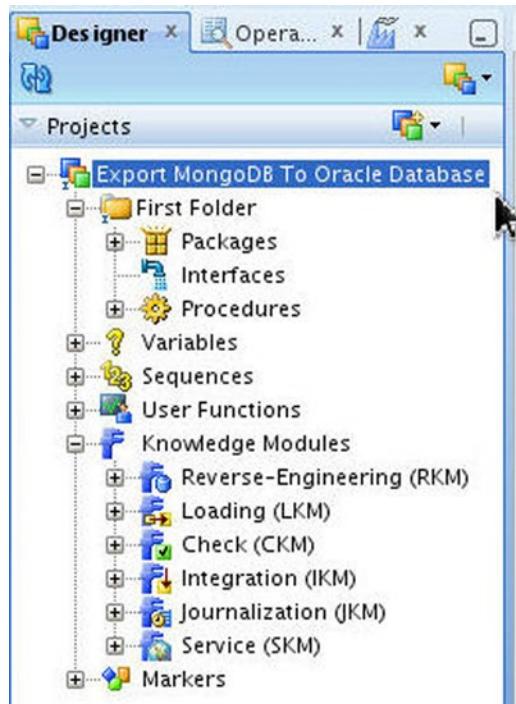
**Figure 12-56.** Selecting Projects ► New Project

2. In the Project Definition specify a Name and click on Save as shown in Figure 12-57.



**Figure 12-57.** Configuring an Integration Project

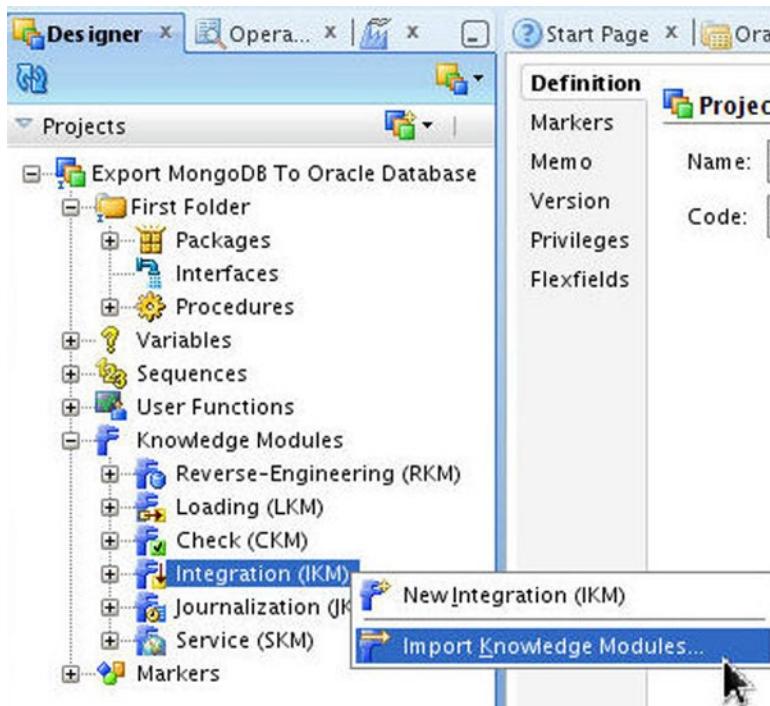
An integration project gets added to Projects as shown in Figure 12-58.



**Figure 12-58.** New Integration Project

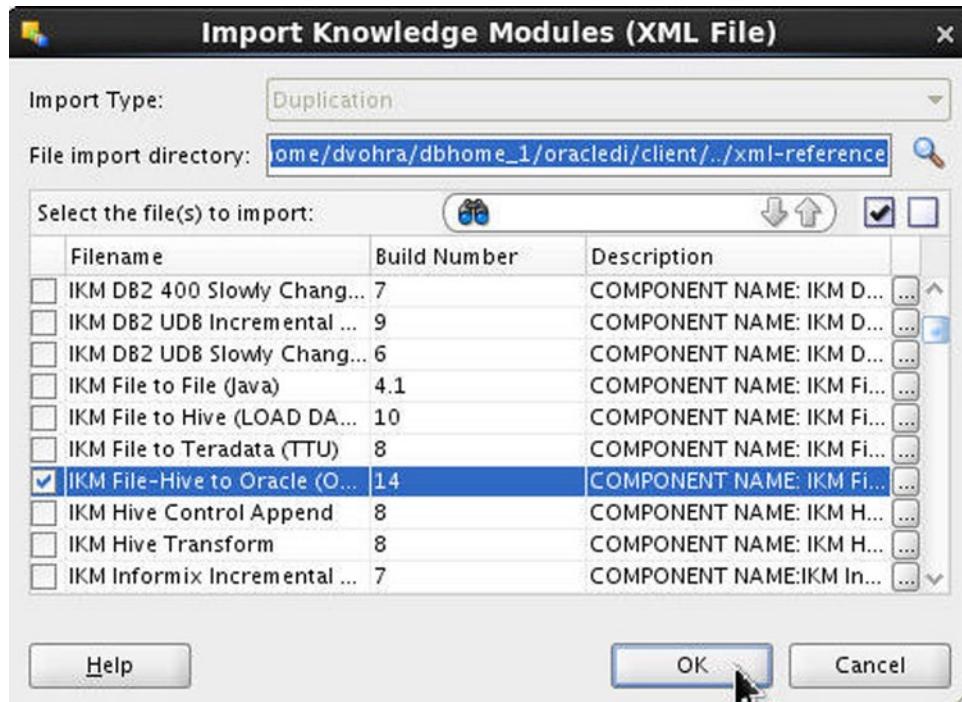
We shall be using the File-Hive to Oracle integration knowledge module for integrating the Hive table data to Oracle Database. We need to add the IKM File-Hive to Oracle knowledge module to the project.

3. Right-click on Knowledge Modules ➤ Integration and select Import Knowledge Modules as shown in Figure 12-59.



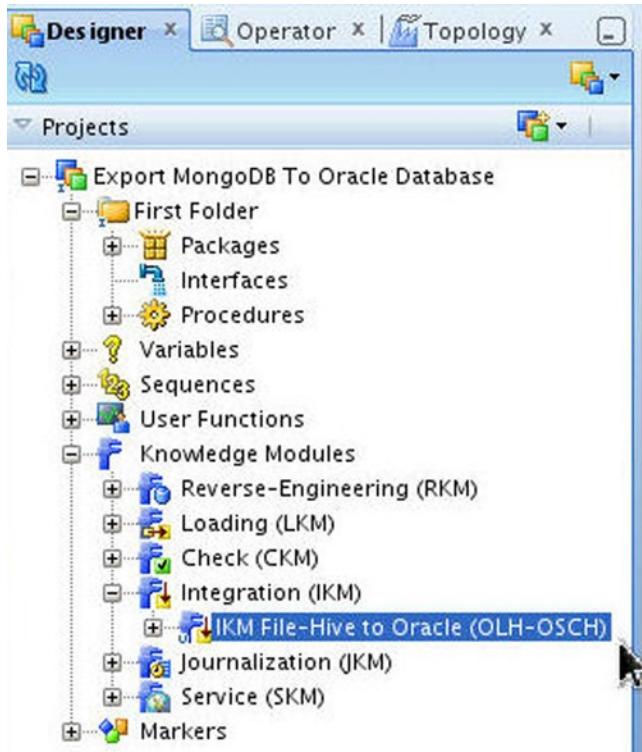
**Figure 12-59.** Selecting Knowledge Modules ➤ Integration ➤ Import Knowledge Modules

- Select the IKM File-Hive to Oracle knowledge module in Import Knowledge Modules as shown in Figure 12-60. With IKM File-Hive to Oracle knowledge, module source could be a file or Hive and its target is Oracle Database. Click on OK.



**Figure 12-60.** Selecting the IKM File-Hive to Oracle Knowledge Module

5. Click on Close in the Import Report. The IKM File-Hive to Oracle knowledge module gets added to the Integration Knowledge Modules as shown in Figure 12-61.

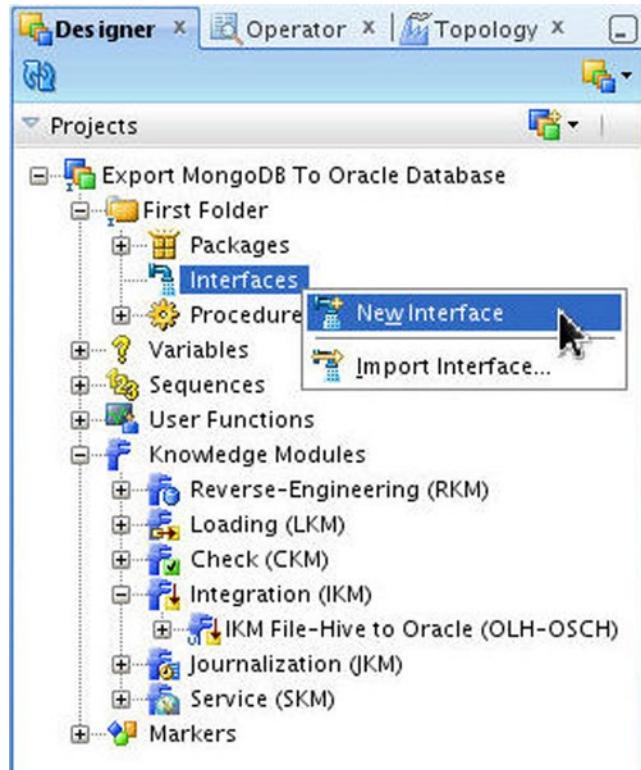


**Figure 12-61.** Knowledge Module IKM File-Hive to Oracle

## Creating the Integration Interface

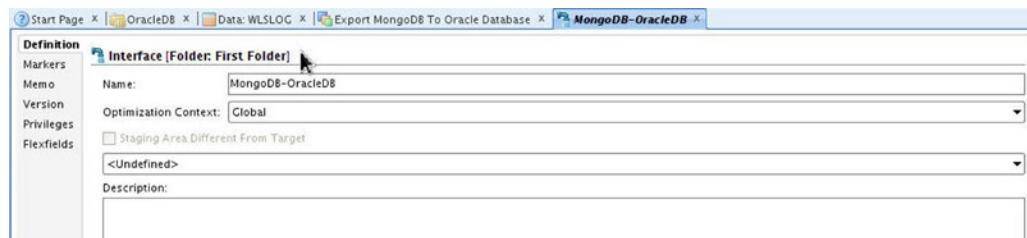
An integration interface defines the mapping of the source datastores to the target datastores including the flow of data and the knowledge module used in the integration.

1. Right-click on First Folder ➤ Interfaces within the integration project and select New Interface as shown in Figure 12-62.



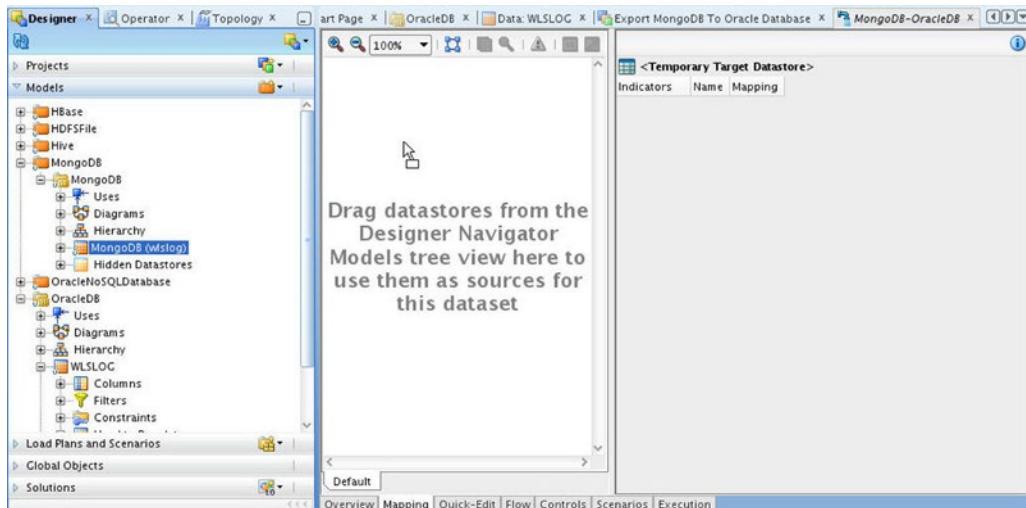
**Figure 12-62.** Selecting Interfaces ➤ New Interface

2. In the Interface Definition specify a Name and select Optimization Context as Global as shown in Figure 12-63.



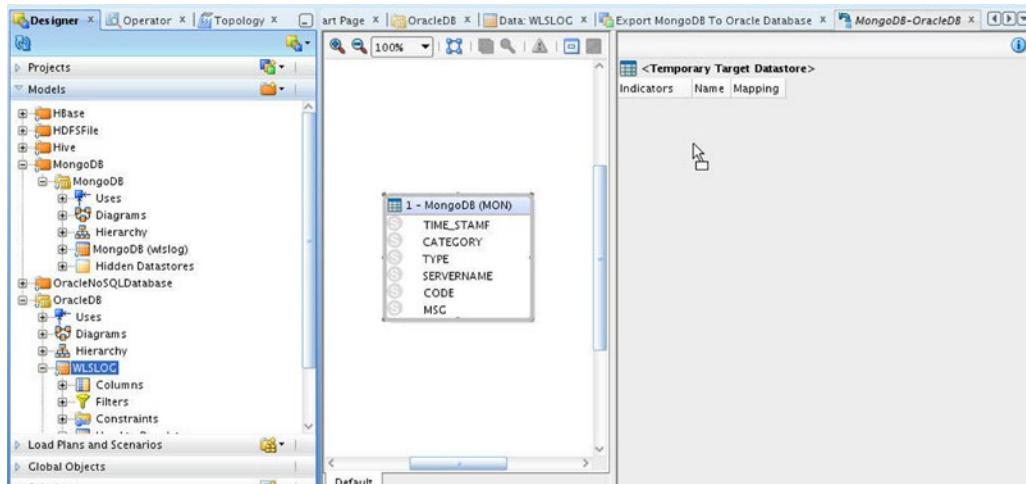
**Figure 12-63.** Configuring an Interface

- Select the Mapping tab. Select the MongoDB (wlslog) datastore from the MongoDB model and drag and drop the datastore in the region for source datastores as shown in Figure 12-64.



**Figure 12-64.** Adding the MongoDB Datastore to the Mapping

- The MongoDB datastore gets added to the source datastores. Similarly select the WLSLOG datastore for Oracle Database model OracleDB and drag and drop the datastore in the region for the target datastore as shown in Figure 12-65.



**Figure 12-65.** Adding the Oracle Database Datastore to the Mapping

5. In the Automap dialog click on Yes to perform automatic mapping as shown in Figure 12-66.



**Figure 12-66.** Automap Dialog

The source and target datastores get defined for the mapping in the integration interface, and the datastore diagrams get added as shown in Figure 12-67.

A screenshot of the Oracle Data Integrator interface. On the left, there's a tree view under '1 - MongoDB (MC)' showing fields: TIME\_STAMP, CATEGORY, TYPE, SERVERNAME, CODE, and MSG, all marked with a red 'S'. On the right, a table titled 'Target Datastore - WLSLOG' shows mappings between MongoDB fields and Oracle database columns. The table has columns: Position, Indicators, Name, and Mapping. The data is as follows:

Position	Indicators	Name	Mapping
1	Y	TIME_STAMP	MON.TIME_STAMP
2	Y	CATEGORY	MON.CATEGORY
3	Y	TYPE	MON.TYPE
4	Y	SERVERTNAME	MON.SERVERNAME
5	Y	CODE	MON.CODE
6	Y	MSG	MON.MSG

**Figure 12-67.** Source and Target Datastores

- Select the Quick-Edit tab and select Staging Area in the Execute On column for the target datastore WLSLOG as shown in Figure 12-68.

The screenshot shows the Oracle Data Integrator interface with the following details:

- Sources:** A table showing a single source named "MongoDB" with alias "MON".
- Mappings:** A table mapping source columns to target columns in the "WLSLOG" target database. The columns are:
 

Indicator	UK	CN	Target Column Name	Data Type	Length	Scale	Mapping Expression	Execute On	Insert	Update	UD1	UD2	UD3	UD4
			TIME_STAMP	VARCHAR2	255	0	MON_TIME_STAMP	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			CATEGORY	VARCHAR2	255	0	MON_CATEGORY	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			TYPE	VARCHAR2	255	0	MON_TYPE	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			SERVERNAME	VARCHAR2	255	0	MON_SERVERNAME	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			CODE	VARCHAR2	255	0	MON_CODE	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			MSG	VARCHAR2	255	0	MON.MSG	Staging Area	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 12-68. Configuring Target Datastore WLSLOG

- Select the Flow tab. The flow diagram describes the flow of data from the source datastore to the target datastore. The default flow diagram has the staging area in the target database. But, the staging area should be in the source database as shown in Figure 12-69.

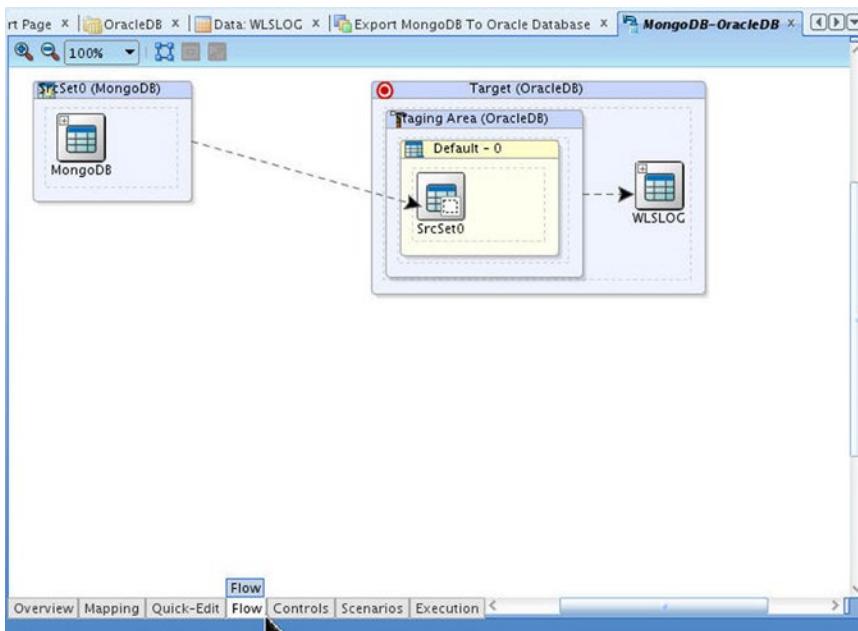
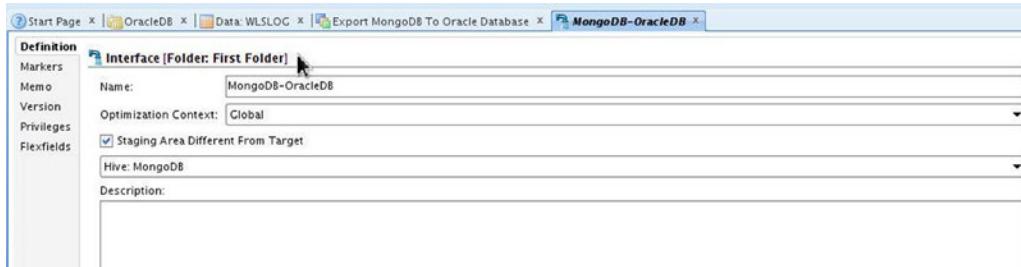


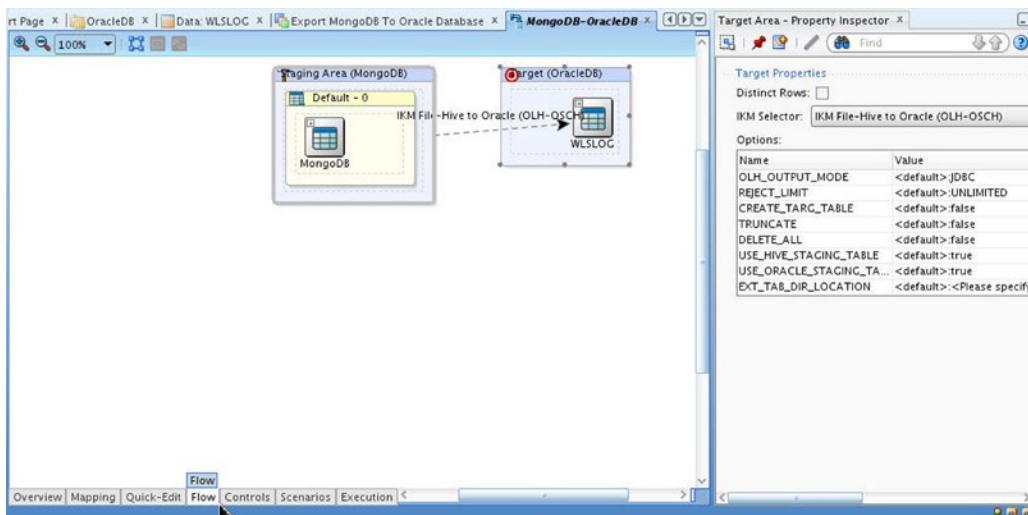
Figure 12-69. Default Flow Diagram

- Select the Overview tab and select Staging Area Different From Target. Even if the check box is already selected, deselect and select again. Select the staging area as Hive:MongoDB as shown in Figure 12-70.



**Figure 12-70.** Configuring Staging Area

The Flow diagram gets updated to indicate flow of data from a staging area in the source database to the target database as shown in Figure 12-71. The IKM Selector should be selected as IKM File-Hive to Oracle.



**Figure 12-71.** Configured Flow Diagram

9. Click on Save to save the integration interface as shown in Figure 12-72.

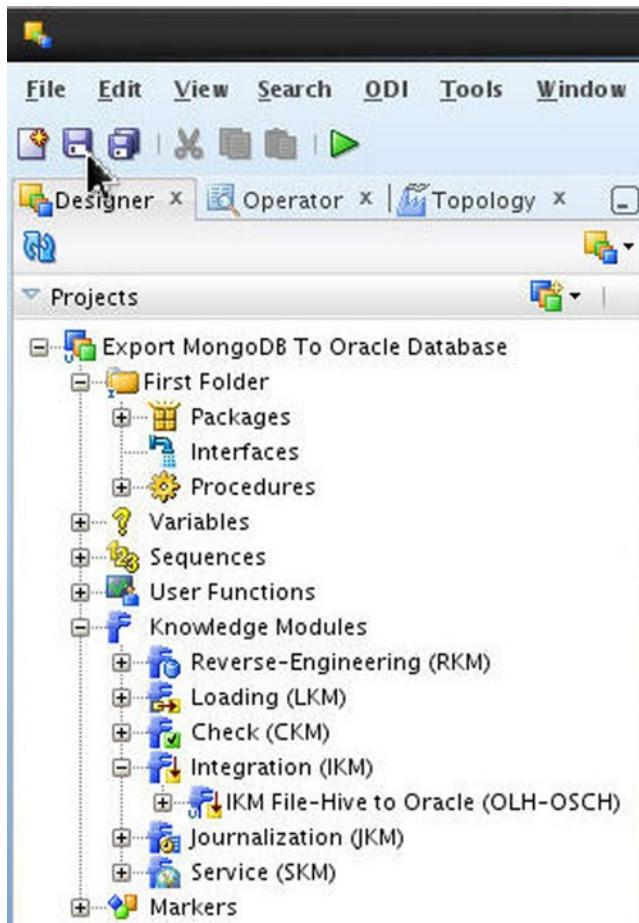
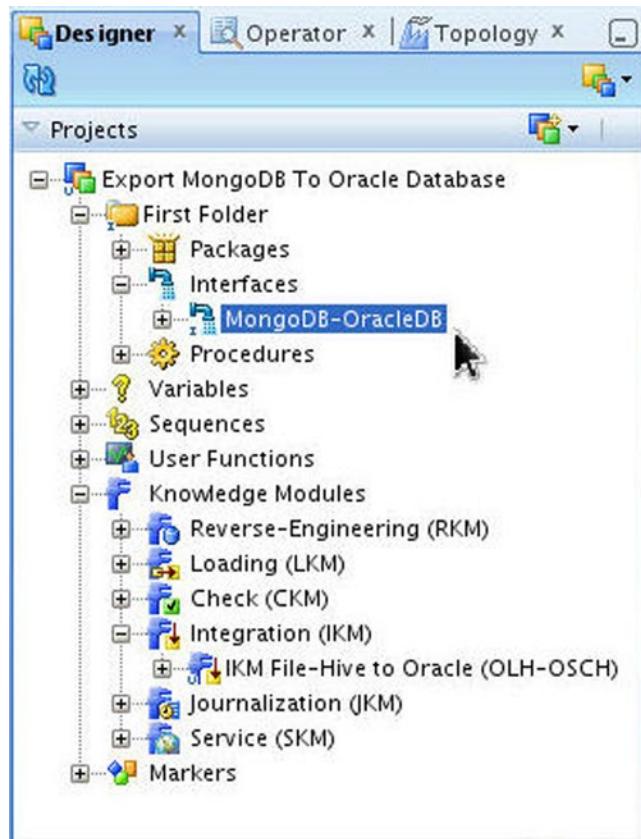


Figure 12-72. Saving Interface

An integration interface gets added as shown in Figure 12-73.

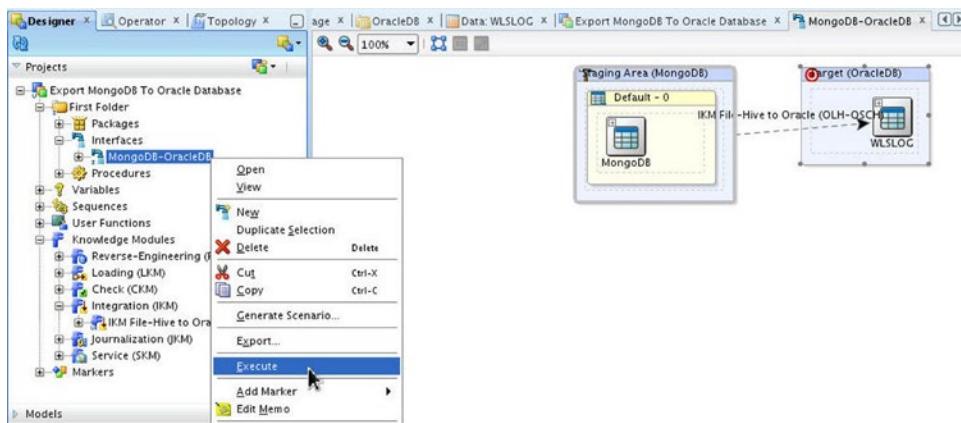


**Figure 12-73.** New Integration Interface

## Running the Interface

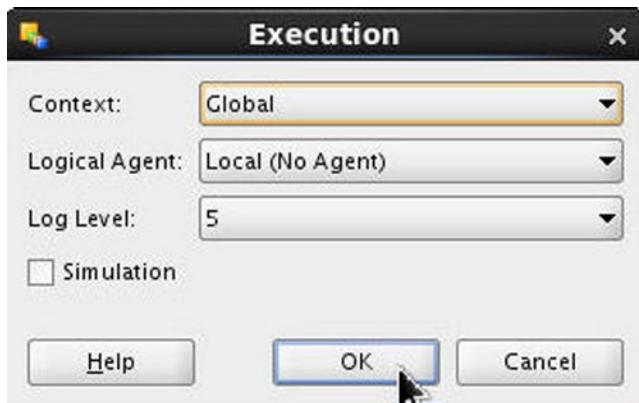
In this section we shall run the integration interface to integrate the data from MongoDB to Oracle Database.

1. Right-click on the integration interface node and select Execute as shown in Figure 12-74.



**Figure 12-74.** Running the Interface

2. In Execution select OK as shown in Figure 12-75.



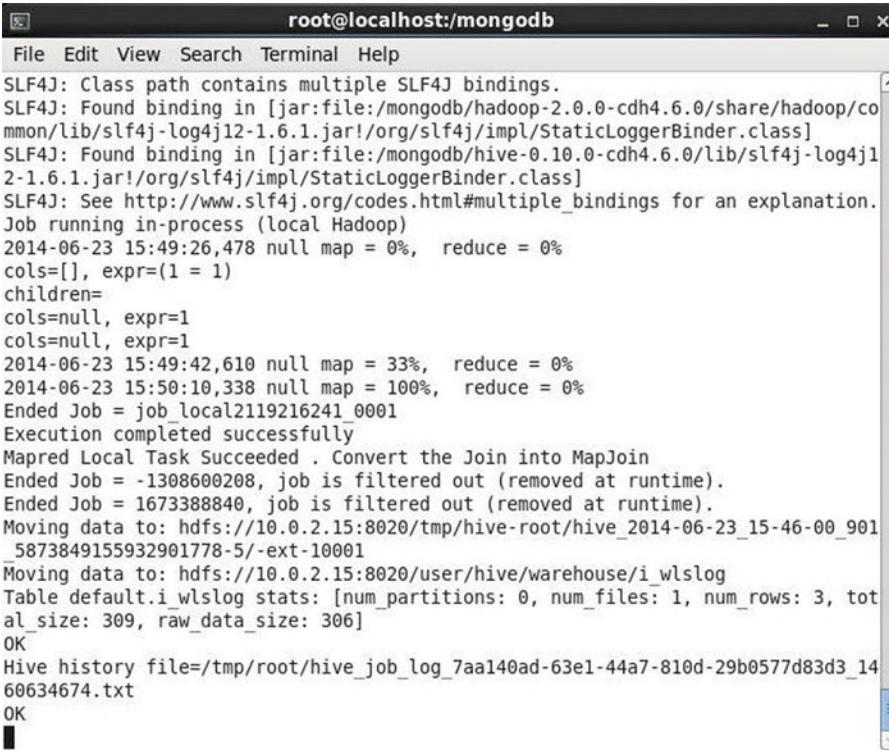
**Figure 12-75.** Configuring the Interface Run

3. A Session started dialog gets displayed. Click on OK as shown in Figure 12-76.



**Figure 12-76.** Information Dialog for Integration Session Started

Three MapReduce jobs run to integrate Hive table data into Oracle Database as shown in Figure 12-77.



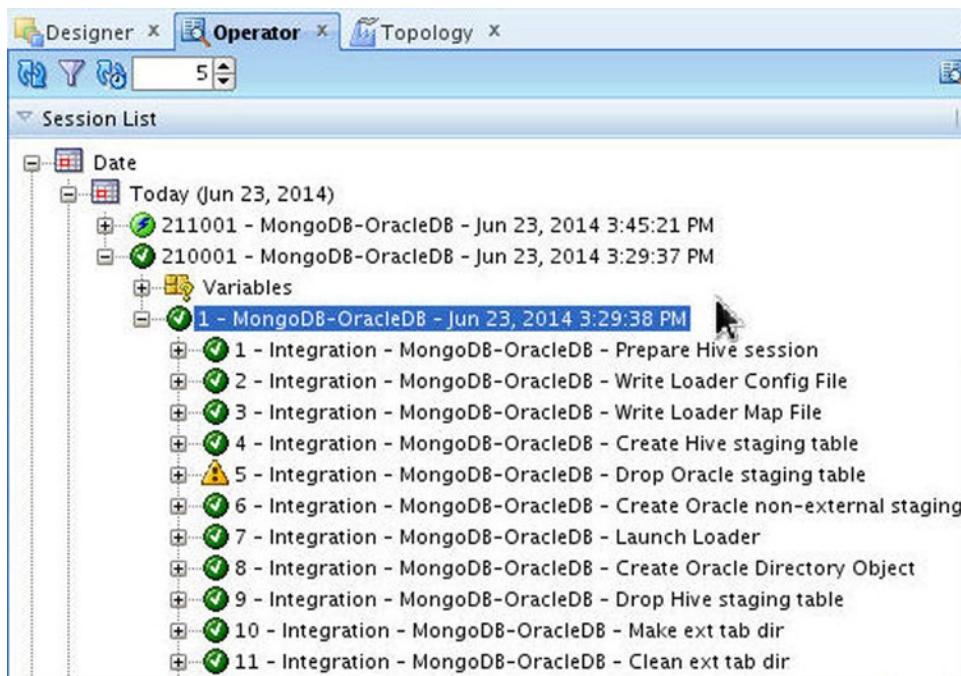
```

root@localhost:/mongodb
File Edit View Search Terminal Help
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/mongodb/hadoop-2.0.0-cdh4.6.0/share/hadoop/co
mmon/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/mongodb/hive-0.10.0-cdh4.6.0/lib/slf4j-log4j1
2-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
Job running in-process (local Hadoop)
2014-06-23 15:49:26,478 null map = 0%, reduce = 0%
cols=[], expr=(1 = 1)
children=
cols=null, expr=1
cols=null, expr=1
2014-06-23 15:49:42,610 null map = 33%, reduce = 0%
2014-06-23 15:50:10,338 null map = 100%, reduce = 0%
Ended Job = job_local2119216241_0001
Execution completed successfully
Mapred Local Task Succeeded . Convert the Join into MapJoin
Ended Job = -1308600208, job is filtered out (removed at runtime).
Ended Job = 1673388840, job is filtered out (removed at runtime).
Moving data to: hdfs://10.0.2.15:8020/tmp/hive-root/hive_2014-06-23_15-46-00_901
_5873849155932901778-5-ext-10001
Moving data to: hdfs://10.0.2.15:8020/user/hive/warehouse/i_wlslog
Table default.i_wlslog stats: [num_partitions: 0, num_files: 1, num_rows: 3, tot
al_size: 309, raw_data_size: 306]
OK
Hive history file=/tmp/root/hive_job_log_7aa140ad-63e1-44a7-810d-29b0577d83d3_14
60634674.txt
OK

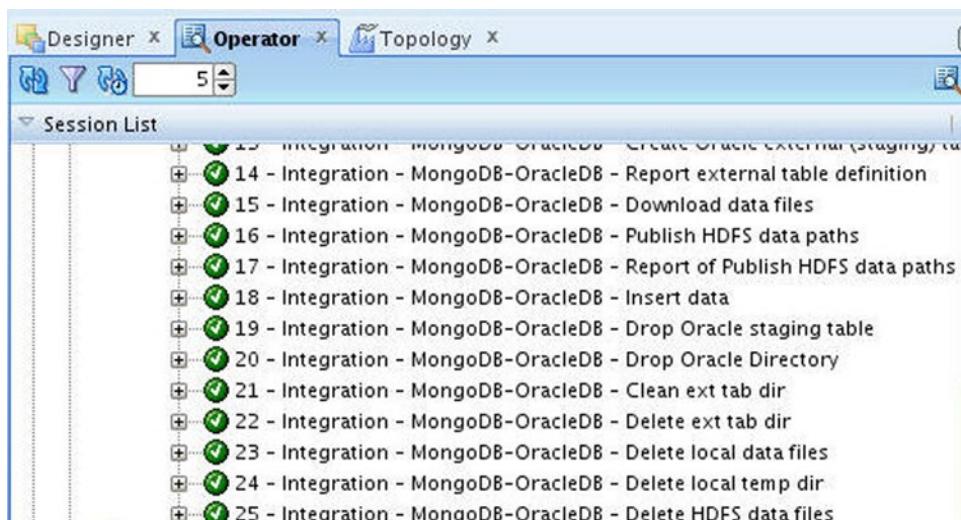
```

**Figure 12-77.** Output from Integrating MongoDB based Hive Table into Oracle Database

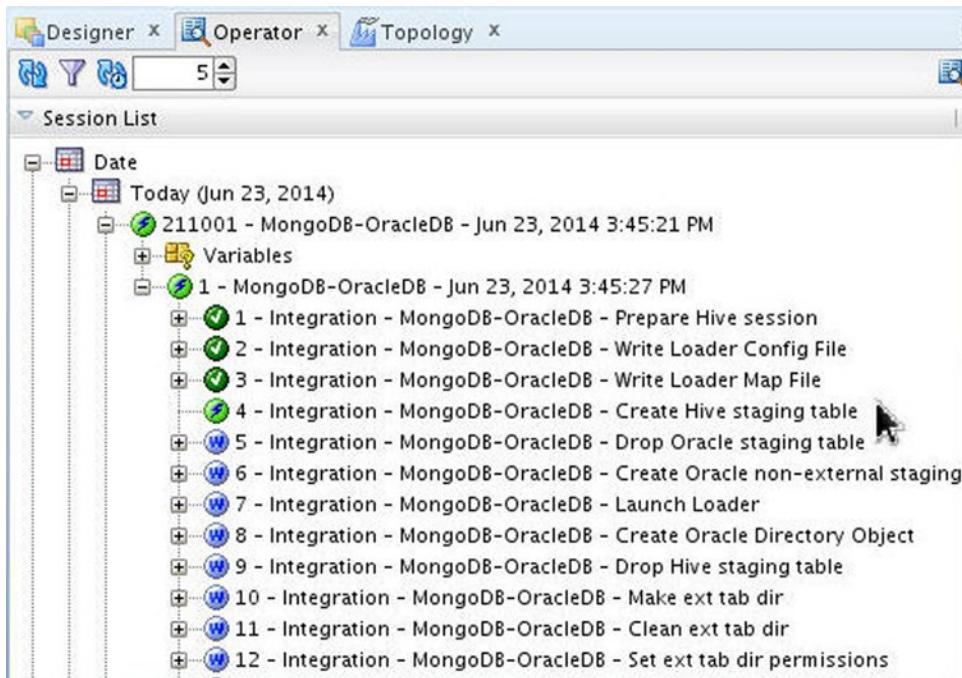
4. Select the Operator tab. Select the session from the Session List. The different phases of the integration are listed, including creating staging tables and launching Oracle Loader for Hadoop as shown in Figure 12-78.

**Figure 12-78.** Integration Phases

After data has been integrated staging tables, Oracle Directory, ext tab dir, local data files, local temp files, and HDFS data files get deleted as shown in Figure 12-79.

**Figure 12-79.** Integration Phases for Deleting Files

One of the advantages of using Oracle Data Integrator over using Oracle Loader for Hadoop on the command line is that the integration gets suspended, without getting terminated, if some configured parameter is not found. For example, if the target database table OE.WLSLOG is not found the integration gets suspended, as in the Create Hive staging table phase as shown in Figure 12-80.



**Figure 12-80.** Suspended Integration Phase

- The session is still running and gets completed if the configuration parameter is fixed, for example, add a missing schema for a database table. Alternatively the integration session may be stopped. Right-click on the session and select one of the Stop options to terminate the session as shown in Figure 12-81.

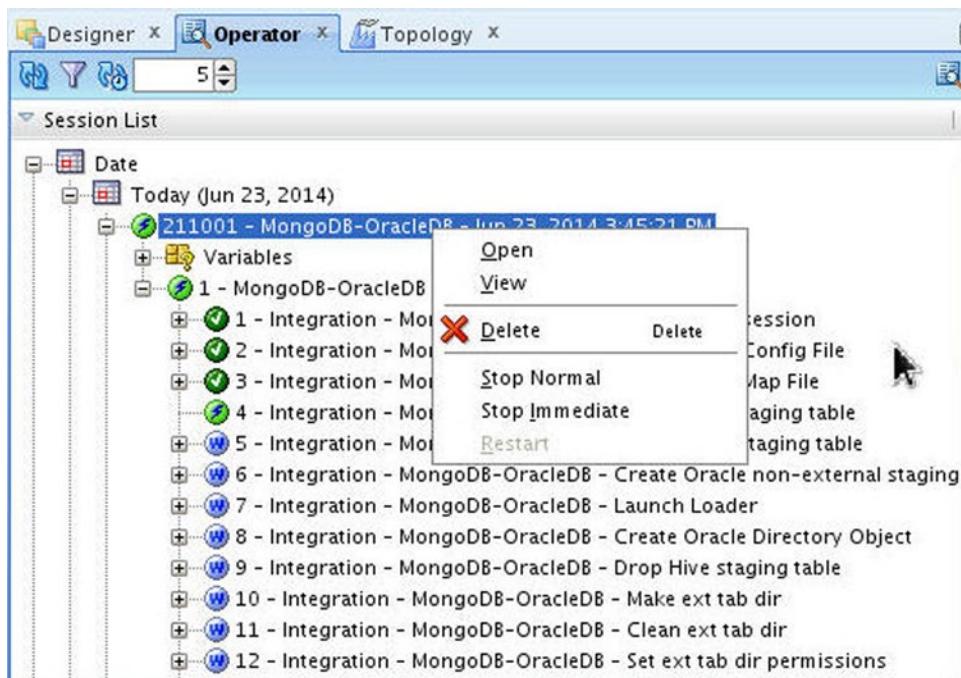
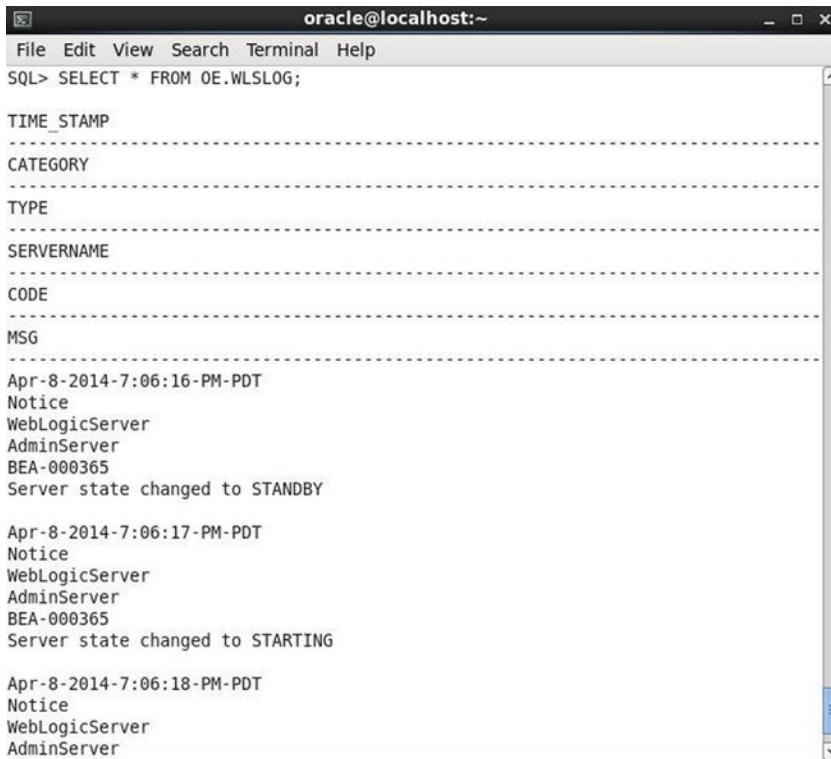


Figure 12-81. Options to Stop Integration Session

## Selecting Integrated Data in Oracle Database Table

Having integrated MongoDB datastore into Oracle Database, run a SELECT statement in SQL/Plus to select-list the integrated data as shown in Figure 12-82.



```
oracle@localhost:~$ File Edit View Search Terminal Help
SQL> SELECT * FROM OE.WLSLOG;
TIME_STAMP

CATEGORY

TYPE

SERVERNAME

CODE

MSG

Apr-8-2014-7:06:16-PM-PDT
Notice
WebLogicServer
AdminServer
BEA-000365
Server state changed to STANDBY

Apr-8-2014-7:06:17-PM-PDT
Notice
WebLogicServer
AdminServer
BEA-000365
Server state changed to STARTING

Apr-8-2014-7:06:18-PM-PDT
Notice
WebLogicServer
AdminServer
```

**Figure 12-82.** Selecting Integrated Data

## Summary

In this chapter we integrated MongoDB data into Oracle Database in Oracle Data Integrator using the IKM File-Hive to Oracle knowledge module.

This chapter concludes the book. You have learned about the different aspects of development with MongoDB such as accessing MongoDB with Java, PHP, Ruby, and Node.js. We also discussed migrating some of the other NoSQL databases - Apache Cassandra and Couchbase - to MongoDB. We learned about using Java EE frameworks Spring Data and Kundera with MongoDB. We also introduced using Hadoop and Oracle Data Integrator with MongoDB.

# Index

## A

Apache Cassandra documents  
catalog table list, 263  
CQL 3, 257  
CreateCassandraDatabase class, 258–260  
Datastax Java driver, 255  
datastax keyspace, 261–262  
Maven project in Eclipse IDE  
    configuration, 247–248  
    CreateCassandraDatabase class  
        configuration, 251  
        dependencies, 253–254  
        Java Class selection, 250  
    MigrateCassandraToMongoDB  
        class configuration, 252  
        selection, 246  
MigrateCassandraToMongoDB class, 263  
partition key, 257  
Session class methods, 256  
software installation, 243  
start up, 244–245  
strategy class, 257  
Apache Hadoop, 405  
Apache Hive  
    data store (*see* Data store)  
    definition, 405  
    environment setting, 406  
    external table  
        adding new user, 423  
        adding wlslog table, 425  
        creation, 425  
        Hive thrift server, 424  
        properties, 424  
        shell commands, 422  
    storage handler, 405  
App.java methods  
    addDocument(), 358  
    addDocumentBatch(), 361  
    createCollection(), 356  
    find(), 370

    findAll(), 368  
    findById(), 363  
    findOne(), 364  
    invocations and code sections, 383  
    removing documents, 379  
    updateFirst(), 373  
    updateMulti(), 376

## B

    BSON document  
        creation  
            append() method, 13  
            constructors, 11  
            CreateMongoDBDocument class, 14  
            CreateMongoDBDocument.java  
                application, 11  
                document class constructors, 12  
                document class utility methods, 12  
            find() method, 13  
            get(String key) method, 13  
            iterator() method, 13  
            keySet() method, 13  
            listCollectionNames() method, 12  
            MongoClient instance creation, 11  
            MongoCollection<Document>instance, 12  
            MongoCollection  
                <TDocument> interface, 13  
                next() method, 13  
                output storage, 15  
        model class  
            append(String key,  
                Object value) method, 19  
            BasicDBObject class, 17–18  
            CreateMongoDBDocument  
                Model class, 18, 20  
                fields, 17  
                FindIterable<TDокумент>, 19  
                getCollection() Method, 18  
                getCollection  
                    (String collectionName) method, 18

## ■ INDEX

[BSON document \(cont.\)](#)

[insertMany\(\) Method, 16, 19](#)  
        [iterator\(\) method, 19](#)  
        [MongoClient instance, 18](#)  
        [MongoDatabase instance, 18](#)  
        [Serializable interface, 17](#)

## ■ C

[content\(\) method, 293](#)

[Couchbase and MongoDB](#)

[catalog1 ID JSON document, 287](#)  
        [catalog2 Document JSON, 288](#)  
        [catalog\\_view, 290](#)  
        [couchbase server community, 273](#)  
        [CreateCouchbaseDocument.java Application, 286](#)  
        [development view, 289](#)  
        [empty couchbase bucket, 274](#)  
        [function maps, 288](#)  
        [getString\(String fieldName\) method, 293](#)  
        [insertOne\(TDocument document\) method, 293](#)

[jar files, 283](#)

[Java Class, 278](#)

[Java Class Wizard, 280](#)  
        [Java ▶ Java Class, 279](#)  
        [Package Explorer, 281](#)

[JsonObject class, 284](#)

[map function, 291](#)

[Maven dependencies, 281](#)

[Maven ▶ Maven project, 275–276](#)

[Maven project artifacts, 277](#)

[MigrateCouchbaseToMongoDB application, 292, 294–295](#)

[Mongo Shell, 295](#)

[openBucket\(\) method, 283](#)

[Package Explorer, 278](#)

[createEntityManager\(\) method, 328](#)

[create\(\) methods, 283](#)

[CRUD operations. See App.java methods](#)

## ■ D

[Data store](#)

[adding objects, 421](#)  
        [command, 410](#)  
        [creating java project MongoDB, 414](#)  
        [eclipse installation, 410](#)  
        [java class creation, 418](#)  
        [java driver, 416](#)  
        [java project creation, 412, 419](#)  
        [MongoClient class, 419](#)  
        [output folder, 412](#)  
        [property selection, 415](#)  
        [running java application, 421](#)

[selecting java class, 417](#)

[selecting java project, 411](#)

[source folder, 413](#)

[delete\(\) method, 342](#)

[disconnect\(\) method, 284](#)

[Documents](#)

[bulkWrite method, 239](#)  
        [catalog collection, 205](#)  
        [createCollection\(\) method, 213](#)  
        [cursor class methods, 218](#)  
        [deleteMany\(\) Method, 237](#)  
        [findAndModify\(\) Method, 222](#)  
        [findAndRemove\(\) Method, 225](#)  
        [findOneAndDelete\(\) Method, 235](#)  
        [findOneAndReplace\(\) Method, 227](#)  
        [findOneAndUpdate\(\) Method, 229](#)  
        [findOne\(\) Method, 211](#)  
        [insertMany method, 208](#)  
        [insertOne\(\) Method, 206](#)  
        [subset documents, 215](#)  
        [updateMany\(\) Method, 233](#)

## ■ E, F

[emit\(\) function, 288](#)

[empty\(\) method, 284](#)

## ■ G, H, I, J

[getCollection\(String collectionName\) method, 12](#)  
    [getResultSet\(\) method, 329](#)

## ■ K, L

[Kundera-mongo module, 305](#)

[Client Class KunderaClient, 326](#)

[CRUD operations, 325, 327](#)

[DELETE statement, 331](#)

[EntityManager class, 328](#)

[EntityManagerFactory object, 328](#)

[executeUpdate\(\) method, 330](#)

[JPA query, 329](#)

[KunderaClient class, 330](#)

[persist\(\) method, 330](#)

[exec-maven-plugin, 311](#)

[Java EE developers, 306](#)

[JPA client class, 331–332](#)

[db.catalog.find\(\) method, 339](#)

[delete\(\) Method, 343](#)

[exec-maven-plugin, 338](#)

[findByClass\(\) method, 339](#)

[findByClass\(\) Method, 340](#)

[KunderaClient class methods, 339](#)

[Maven Project installation, 333–334](#)

[pom.xml ▶ Run As ▶ Run Configuration, 336](#)

update() method, 341  
 update() method, 342  
 JPA object/relational mapping, 312  
   JPA entity class, 316  
     entity class catalog, 314–315  
     Java ► Class, 313  
     JPA entity class, 317  
 kundera-mongo library, 307  
 KunderaMongoDB, 310  
 maven-compiler-plugin, 311  
 Maven ► Maven Project, 307  
 maven project directory structure, 327  
 maven project wizard, 308–309  
 META-INF folder, 321–322, 324  
 MongoDB Collection, 306  
 MongoDB-specific configuration file, 320  
 persistence properties, 318–319  
 persistence.xml file, 325

## M

map() function, 288  
 MongoDB server  
   acquiring data, 22  
   BSON document. *see* BSON document  
   data deletion, 32  
   data updation  
     document verification, 29  
     replaceOne() method, 28  
   insertOne  
     (TDocument document) method, 27  
 MongoClient instance, 27  
 MongoCollection  
   <TD**oc**ument> Methods, 26  
 output, 31  
 replaceOne() method, 32  
 running, 30  
 updateMany(Bson filter,  
   Bson update) method, 28  
 updateOne() method, 32  
 updateOne(Bson filter,  
   Bson update) method, 28  
 update operators, 27  
 Maven project  
   configuration, 4  
   Java Build Path, 9  
   Java classes, 8  
   Java ► Class selection, 6  
   Mongo Java Driver  
     Dependency Configuration, 10  
     new Class creation, 7  
     pom.xml, 9  
     selection, 3  
 setting up, 1

Mongoimport tool, 297  
 Mongo shell  
   collection  
     capped, 42, 49  
     creation, 48  
     db.collection.drop() method, 52  
     list, 48  
   connect() method, 42  
   database commands, 39, 42  
   databases  
     db.dropDatabase() method, 47  
     getting information, 44  
     instance creation, 46  
   db.createCollection() Helper method, 43  
   db.runCommand() Helper method, 43  
   documents  
     adding an array, 56  
     BSON, 55  
     cursor, 80  
     db.collection.remove() method, 85  
     db.collection.save() method, 60  
     db.collection.update() method, 68  
     duplicate key error, 56  
     findAndModify() method, 82  
     finding selected fields, 76  
     findOne() method, 74–75  
     insert() helper method, 53–54  
     list, 55  
     multiple documents updation, 71  
   download and install, 39  
   error message, 43  
   getDB() method, 41  
   help methods, 44  
   JavaScript helper methods *vs.* help methods, 42  
   start up, 40

## N

Node.js  
   collection class  
     documents (*see* Documents)  
     methods, 198  
     properties, 198  
   connect() method, 189–191  
   Db class  
     db.js Script, 196–197  
     listCollections() method, 194–195  
     methods, 193  
     options, 192  
     parameters, 191  
     properties, 193  
   driver classes, 177–178  
   MongoClient class methods, 186  
   MongoDB installation, 178

## ■ INDEX

### Node.js (*cont.*)

- Node.js driver installation, 185–186
- Node.js installation, 178
- string options, 188
- URI components, 187

## ■ O

### Oracle database table

- Command Parameters, 301
- creation, 299
- to CSV file, 299–300
- JSON data, 302–303
- mongoimport tool, 297
- software install, 298
- wlslog.csv file to

MongoDB Server, 301

### Oracle Data Integrator (ODI)

- data models
  - adding Hive technology, 449
  - new model folder creation, 448
  - OracleDB schema creation, 454
  - Reverse Engineering function, 455
  - selecting datastore, 450
- integrated table, 476
- integration interface
  - configuration, 464
  - mapping datastores, 465
  - saving, 469
  - selecting new, 464
  - staging area creation, 466

### integration project

- creation, 459
- Import Knowledge
  - Modules function, 461

### logical architecture

- adding Oracle technology, 446
- new schema creation, 444
- selecting Hive technology, 443

### physical architecture

- adding new oracle technology, 439
- adding physical schema, 434
- data servers creation, 431
- selecting Hive technology, 430
- table creation, 436

### running interface

- configuration, 471
- information dialog, 471
- integration phases, 472
- output, 472
- selecting node, 471
- stop integration session, 475
- suspended integration phase, 474
- software setting, 427

## ■ P

### PHP MongoDB Database Driver

- compatibility matrix, 94
- Core Classes, 90
- createCollection Method, 99–100
- creating connection, 94
- database info, 97
- documents
  - addDocumentBatch.php script, 108, 110
  - addDocument
    - Exception.php script, 105–106
  - addDocument.php Script, 104–105
  - addMultiple
    - Documents.php Script, 106–107
  - findAllDocuments.php script, 112–114
  - findDocumentSet.php script, 114, 116–117
  - insert method, 103
  - MongoCollection\::findOne()
    - method, 110–112
  - MongoCollection\::remove()
    - method, 125–127
  - MongoCollection\::save()
    - method, 122–124
  - MongoCollection\::update()
    - method, 117–118, 120
  - multiple documents updation, 120–122
- download and install, 91
- exceptions, 90
- installation, 92
- MongoCollection\::drop() method, 101
  - running PHP script, 93

### Plain old Java objects (POJOs), 351

## ■ Q

### query() methods, 292, 331

## ■ R

- rows() method, 292
- Ruby MongoDB driver, 129
  - Collection class constructor, 143, 145
  - Collection Class
    - Instance Attributes, 143
  - Collection Class Instance Methods, 143
  - collection.rb script, 146–147
  - Core Classes, 129–130
  - creating connection
    - Class Attributes, 136
    - Class Constructor Options, 137
    - Class Methods, 136
    - connection.rb script, 137–138
    - Constructor Parameters, 137

Database class constructor, 139–140  
 Database Class Instance Attributes, 139  
 Database Class Methods, 139  
 db.rb script, 140, 142  
 DevKit installation, 133–134  
 documents  
   addDocument.rb script, 148–150  
   bulk operations, 173  
   bulk.rb script, 174–176  
   deleteDocument.rb script, 169  
   findDocument.rb script, 156–157  
   findDocuments.rb script, 159–160  
   find() method, 158, 160  
   insert\_one(document, options = {})  
     method, 147  
   Mongo, 161  
   multiple documents added, 151  
   OperationFailure Error, 151  
   updateDocument.rb script, 162–167, 169  
 download and install, 131  
 installation, 131, 134–135  
 Rubygems  
   installation, 133  
   updation, 133  
 software installation, 131

## ■ S

Spring data  
 App.java (*see* App.java methods)  
 createCatalogInstances(), 358  
 creating new maven project, 346  
 getBean method, 355  
 installation, 350  
 JavaConfig, 351  
 model creation, 353  
 repositories  
   basePackageClasses, 389  
   CatalogService, 390  
   count() method, 391  
   deleteAll() method, 400  
   deleteById() method, 399  
   deleting entities, 399  
   finding entities, 392  
   saving entities, 395  
   SpringMongoApplicationConfig, 389  
 setting up environment, 345

## ■ T, U, V, W, X, Y, Z

Target datastore, 436