

CONTENTS

- » Configuration Options
- » Using the Shell
- » Diagnosing What's Happening
- » Quick Rules
- » Query Operators
- » And more...

MongoDB

By Vlad Mihalcea

MongoDB is a document-oriented database that is easy to use from almost any language. As of August 2014, MongoDB is by far the most popular NoSQL database according to <http://db-engines.com/en/ranking>. This Refcard is intended to help you get the most out of MongoDB and assumes that you already know the basics. If you're just starting out, try these resources:

- For installation notes, see <http://docs.mongodb.org/manual/installation/>
- For a quick tutorial on basic operations, see <http://docs.mongodb.org/manual/tutorial/getting-started/>

CONFIGURATION OPTIONS

SETTING OPTIONS

Startup options for MongoDB can be set on the command line or in a configuration file. The syntax is slightly different between the two. Here are the three types of options:

COMMAND-LINE	CONFIG FILE
<code>--dbpath /path/to/db</code>	<code>dbpath=/path/to/db</code>
<code>--auth</code>	<code>auth=true</code>
<code>-vvv</code>	<code>vvv=true</code>

Run `mongod --help` for a full list of options. Here are some of the most useful:

OPTION	DESCRIPTION
<code>--config /path/to/config</code>	Specifies config file where other options are set.
<code>--dbpath /path/to/data</code>	Path to data directory.
<code>--port 27017</code>	Port for MongoDB to listen on.
<code>--logpath /path/to/file.log</code>	Where the log is stored. This is a path to the exact file, not a directory.
<code>--logappend</code>	On restart, appends to (does not truncate) the existing log file. Always use this when using the <code>--logpath</code> option.
<code>--fork</code>	Forks the mongod as a daemon process.
<code>--auth</code>	Enables authentication on a single server.
<code>--keyFile /path/to/key.txt</code>	Enables authentication on replica sets and sharding. Takes a path to a shared secret key.
<code>--nohttpinterface</code>	Turns off HTTP interface.
<code>--bind_ip address</code>	Only allows connections from the specified network interface.

SEEING OPTIONS

If you started `mongod` with a bunch of options six months ago, how can you see which options you used? The shell has a helper:

```
> db.serverCmdLineOpts()
{ "argv" : [ "./mongod", "--port", "30000" ], "parsed" : { },
  "ok" : 1 }
```

The `parsed` field is a list of arguments read from a `config` file.

USING THE SHELL

SHELL HELP

There are a number of functions that give you a little help if you forget a command:

```
> // basic help
> help
      db.help()           help on db methods
      db.mycoll.help()   help on collection methods
      sh.help()          sharding helpers
      rs.help()          replica set helpers
      help admin         administrative help
      help connect      connecting to a db help
      ...
```

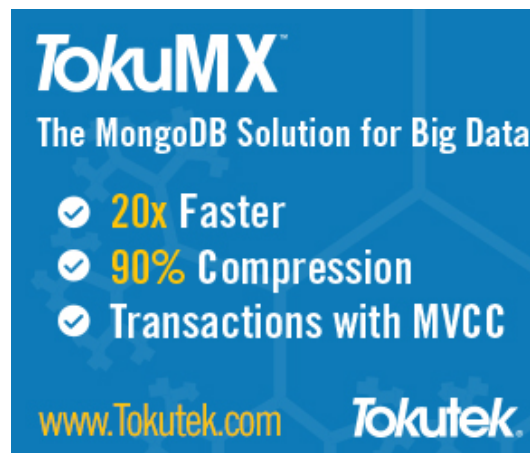
Note that there are separate help functions for databases, collections, replica sets, sharding, administration, and more. Although not listed explicitly, there is also help for cursors:

```
> // list common cursor functions
> db.foo.find().help()
```

You can use these functions and helpers as built-in cheat sheets.

Seeing Function Definitions

If you don't understand what a function is doing, you



TokuMX
The MongoDB Solution for Big Data

- ✓ 20x Faster
- ✓ 90% Compression
- ✓ Transactions with MVCC

www.Tokutek.com Tokutek

can run it without the parentheses in the shell to see its source code:

```
> // run the function
> db.serverCmdLineOpts()
{ "argv" : [ "./mongod" ], "parsed" : { }, "ok" : 1 }
> // see its source
> db.serverCmdLineOpts
```

This can be helpful for seeing what arguments it expects or what errors it can throw, as well as how to run it from another language.

Using edit

The shell has limited multi-line support, so it can be difficult to program in. The shell helper `edit` makes this easier, opening up a text editor and allowing you to edit variables from there. For example:

```
> x = function() { /* some function we're going to fill in */ }
> edit x
<opens emacs with the contents of x>
```

Modify the variable in your editor, then save and exit. The variable will be set in the shell.

Either the `EDITOR` environment variable or a MongoDB shell variable `EDITOR` must be set to use `edit`. You can set it in the MongoDB shell as follows:

```
> EDITOR="/usr/bin/emacs"
```

`edit` is not available from JavaScript scripts, only in the interactive shell.

.mongorc.js

If a `.mongorc.js` file exists in your home directory, it will automatically be run on shell startup. Use it to initialize any helper functions you use regularly and remove functions you don't want to accidentally use.

For example, if you would prefer to not have `dropDatabase()` available by default, you could add the following lines to your `.mongorc.js` file:

```
DB.prototype.dropDatabase = function() {
  print("No dropping DBs!");
}
db.dropDatabase = DB.prototype.dropDatabase;
```

The example above will change the `dropDatabase()` helper to only print a message, and not to drop databases.

Note that this technique should not be used for security because a determined user can still drop a database without the helper. However, removing dangerous admin commands can help prevent fat-fingering.

A couple of suggestions for helpers you may want to remove from `.mongorc.js` are:

- `DB.prototype.shutdownServer`
- `DBCollection.prototype.drop`
- `DBCollection.prototype.ensureIndex`

- `DBCollection.prototype.reIndex`
- `DBCollection.prototype.dropIndexes`

CHANGING THE PROMPT

The shell prompt can be customized by setting the prompt variable to a function that returns a string:

```
prompt = function() {
  try {
    db.getLastError();
  }
  catch (e) {
    print(e);
  }
  return (new Date()+"$ ";
}
```

If you set `prompt`, it will be executed each time the prompt is drawn (thus, the example above would give you the time the last operation completed).

Try to include the `db.getLastError()` function call in your prompt. This is included in the default prompt and takes care of server reconnection and returning errors from writes.

Also, always put any code that could throw an exception in a try/catch block. It's annoying to have your prompt turn into an exception!

DIAGNOSING WHAT'S HAPPENING

VIEWING AND KILLING OPERATIONS

You can see current operations with the `currentOp` function:

```
> db.currentOp()
{
  "inprog" : [
    {
      "opid" : 123,
      "active" : false,
      "locktype" : "write",
      "waitingForLock" : false,
      "secs_running" : 200,
      "op" : "query",
      "ns" : "foo.bar",
      "query" : {
        ...
      },
      ...
    },
    ...
  ]
}
```

Using the `opid` field from above, you can kill operations:

```
> db.killOp(123)
```

Not all operations can be killed or will be killed immediately. In general, operations that are waiting for a lock cannot be killed until they acquire the lock.

INDEX USAGE

Use `explain()` to see which index MongoDB is using for a query.

```
> db.foo.find(criteria).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 2,
  "nscanned" : 2,
  "nscannedObjectsAllPlans" : 2,
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    },
  "server" : "ubuntu:27017"
}
```

There are several important fields in the output of `explain()`:

- *n*: the number of results returned.
- *nscanned*: the number of index entries read.
- *nscannedObjects*: the number of docs referred by the index.
- *indexOnly*: if the query never had to touch the collection itself.
- *nYields*: the number of times this query has released the read lock and waited for other operations to go.
- *indexBounds*: when an index is used, this shows the index scan ranges.

TYPES OF CURSORS

A *BasicCursor* means that no index was used. A *BtreeCursor* means a normal index was used. Parallel cursors are used by sharding and geospatial indexes use their own special cursors.

HINTING

Use `hint()` to force a particular index to be used for a query:

```
> db.foo.find().hint({x:1})
```

SYSTEM PROFILING

You can turn on system profiling to see operations currently happening on a database. There is a performance penalty to profiling, but it can help isolate slow queries.

```
> db.setProfilingLevel(2) // profile all operations
> db.setProfilingLevel(1) // profile operations that take longer
than 100ms
> db.setProfilingLevel(1, 500) // profile operations that take
longer than 500ms
> db.setProfilingLevel(0) // turn off profiling
> db.getProfilingLevel(1) // see current profiling setting
```

Profile entries are stored in a capped collection called `system.profile` in the database in which profiling was enabled. Profiling can be turned on and off for each database.

REPLICA SETS

To find replication lag, connect to a secondary and run

this function:

```
> db.printReplicationStatus()
configured oplog size: 2000MB
log length start to end: 23091secs (6.4hrs)
oplog first event time: Fri Aug 10 2012 04:33:03 GMT+0200
(CEST)
oplog last event time: Mon Aug 20 2012 10:56:51 GMT+0200
(CEST)
now: Mon Aug 20 2012 10:56:51 GMT+0200
(CEST)
```

To see a member's view of the entire set, connect to it and run:

```
> rs.status()
```

This command will show you what it thinks the state and status of the other members are.

Running `rs.status()` on a secondary will show you who the secondary is syncing from in the (poorly named) `syncingTo` field.

SHARDING

To see your cluster's metadata (shards, databases, chunks, etc.), run the following function:

```
> db.printShardingStatus()
> db.printShardingStatus(true) // show all chunks
```

You can also connect to the *mongos* and see data about your shards, databases, collections, or chunks by using "use config" and then querying the relevant collections.

```
> use config
switched to db config
> show collections
chunks
databases
lockpings
locks
mongos
settings
shards
system.indexes
version
```

Always connect to a *mongos* to get sharding information. Never connect directly to a *config* server. Never directly write to a *config* server. Always use sharding commands and helpers.

After maintenance, sometimes *mongos* processes that were not actually performing the maintenance will not have an updated version of the *config*. Either bouncing these servers or running the `flushRouterConfig` command is generally a quick fix to this issue.

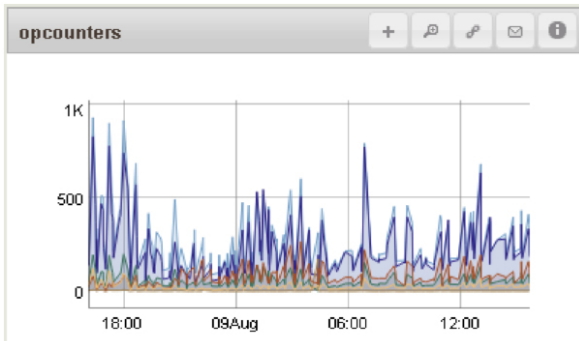
```
> use admin
> db.runCommand({flushRouterConfig:1})
```

Often this problem will manifest as `setShardVersion` failed errors.

Don't worry about `setShardVersion` errors in the logs, but they should not trickle up to your application (you shouldn't get the errors from a driver unless the *mongos* it's connecting to cannot reach any *config* servers).

MONGO MONITORING SERVICE (MMS)

MMS is a free, easily-setup way to monitor MongoDB. To use it, create an account at <http://mms.10gen.com>.



See <http://mms.10gen.com/help> for more documentation.

QUICK RULES

DATABASES

Database names cannot contain ".", "\$", or "\0" (the null character). Names can only contain characters that can be used on your filesystem as filenames. *Admin*, *config*, and *local* are reserved database names (you can store your own data in them, but you should never drop them).

COLLECTIONS

Collection names cannot contain "\$" or "\0". Names prefixed with "system." are reserved by MongoDB and cannot be dropped (even if you created the collection). Dots are often used for organization in collection names, but they have no semantic importance. A collection named "che.se" has no more relation to a collection named "che" than one named "cheese" does.

FIELD NAMES

Field names cannot contain "." nor "\0". Fields should only contain "\$" when they are database references.

INDEX OPTIONS

background	Builds indexes in the background, while other connections can read and write.
unique	Every value for this key must be distinct.
sparse	Non-existent values are not indexed. Very handy for indexing unique fields that some documents might not have.
expireAfterSeconds	Takes a number of seconds and makes this a "time to live" collection.
dropDups	When creating unique indexes, drops duplicate values instead of erroring out. Note that this will delete documents with duplicate values!

QUERY FORMAT

Queries are generally of the form:

```
{key : {$op : value}}
```

For example:

```
{age : {$gte : 18}}
```

There are three exceptions to this rule: \$and, \$or, and \$nor, which are all top-level:

```
{$or : [{age : {$gte : 18}}, {age : {$lt : 18}, parentalConsent:true}]}
```

UPDATE FORMAT

Updates are always of the form:

```
{key : {$mod : value}}
```

For example:

```
{age : {$inc : 1}}
```

QUERY OPERATORS

✓:Matches

X:Does not match

OPERATOR	EXAMPLE QUERY	EXAMPLE DOCS
\$gt, \$gte, \$lt, \$lte, \$ne	{numSold : {\$lt:3}}	✓{numSold: 1} X{numSold: "hello"} X{x : 1}
\$in, \$nin	{age : {\$in : [10, 14, 21]}}	✓{age: 21} ✓{age: [9, 10, 11]} X{age: 9}
\$all	{hand : {\$all : ["10","J","Q","K","A"]}}	✓{hand: ["7", "8", "9", "10", "J", "Q", "K", "A"]} X{hand:["J","Q","K"]}
\$not	{name : {\$not : /jon/i}}	✓{name: "Jon"} X{name: "John"}
\$mod	{age : {\$mod : [10, 0]}}	✓{age: 50} X{age: 42}
\$exists	{phone : {\$exists: true}}	✓{phone: "555-555-5555"} X{phones: ["555-555-5555", "1-800-555-5555"]}
\$type*	{age : {\$type : 2}}	✓{age : "42"} X{age : 42}
\$size	{"top-three":{\$size:3}}	✓{"top-three":["gold","silver","bronze"]} X{"top-three":["blue ribbon"]}

\$regex	{role: /admin.*/i} {role: {\$regex:'admin.*', \$options: 'i'}}}	<ul style="list-style-type: none"> ✓ {role: "administrator"} ✓ {role: "Admin"} X {role: "user"}
---------	---	--

See <http://www.mongodb.org/display/DOCS/Advanced+Queries> for a full list of types.

UPDATE MODIFIERS

MODIFIER	START DOC	EXAMPLE MOD	END DOC
\$set	{x:"foo"}	{\$set:{x:[1,2,3]}}	{x:[1,2,3]}
\$unset	{x:"foo"}	{\$unset:{x:true}}	{}
\$inc	{countdown:5}	{\$inc:{countdown:-1}}	{countdown:4}
\$push, \$pushAll	{votes:[-1,-1]}	{\$push:{votes:-1}}	{votes:[-1,-1,-1]}
\$pull, \$pullAll	{blacklist:["ip1", "ip2", "ip3"]}	{\$pull:{blacklist:"ip2"}}	{blacklist:"ip1", "ip3"}
\$pop	{queue:["1pm", "3pm", "8pm"]}	{\$pop:{queue:-1}}	{queue:["3pm", "8pm"]}
\$addToSet, \$each	{ints:[0,1,3,4]}	{\$addToSet:{ints:{\$each:[1,2,3]}}}	{ints:[0,1,2,3,4]}
\$rename	{nmae:"sam"}	{\$rename:{nmae:"name"}}	{name:"sam"}
\$bit	{permission:6}	{\$bit:{permissions:{or:1}}}	{permission:7}

AGGREGATION PIPELINE OPERATORS

The aggregation framework can be used to perform everything from simple queries to complex aggregations.

To use the aggregation framework, pass the aggregate() function a pipeline of aggregation stages:

```
> db.collection.aggregate({$match:{x:1}},
... {$limit:10},
... {$group:{_id : "$age"}}})
```

A list of available stages:

OPERATOR	DESCRIPTION
{\$project : projection}	Includes, exclude,s renames, and munges fields.
{\$match : match}	Queries, takes an argument identical to that passed to find().
{\$limit : num}	Limits results to num.
{\$skip : skip}	Skips num results.
{\$sort : sort}	Sorts results by the given fields.
{\$group : group}	Groups results using the expressions given (see table below).
{\$unwind : field}	Explodes an embedded array into its own top-level documents.
{\$redact : expression}	Removes sensitive information from the current aggregation result.

{\$out : output-collection}	Writes the aggregation result documents to a specified collection.
-----------------------------	--

To refer to a field, use the syntax *\$fieldName*. For example, this projection would return the existing "time" field with a new name, "time since epoch":

```
{$project: {"time since epoch": "$time"}}
```

\$project and *\$group* can both take expressions, which can use this *\$fieldName* syntax as shown below:

EXPRESSION OP EXAMPLE	DESCRIPTION
\$add : ["\$age", 1]	Adds 1 to the age field.
\$divide : ["\$sum", "\$count"]	Divides the sum field by count.
\$mod : ["\$sum", "\$count"]	The remainder of dividing sum by count.
\$multiply : ["\$mph", 24, 365]	Multiplies mph by 24*365
\$subtract : ["\$price", "\$discount"]	Subtracts discount from price.
\$strcasecmp : ["\$ZZ", "\$name"]	1 if name is less than ZZ, 0 if name is ZZ, -1 if name is greater than ZZ.
\$substr : ["\$phone", 0, 3]	Gets the area code (first three characters) of phone.
\$toLowerCase : "\$str"	Converts str to all lowercase.
\$toUpperCase : "\$str"	Converts str to all uppercase.
\$ifNull : ["\$mightExist", \$add : ["\$doesExist", 1]]	If mightExist is not null, returns mightExist. Otherwise returns the result of the second expression.
\$cond : [exp1, exp2, exp3]	If exp1 evalutes to true, return exp2, otherwise return expr3.
\$let: { vars, in }	Binds variables to be used in subexpressions.
\$map : { input : [1, 2], as : "\$v", in : { \$multiply: ["\$\$v", 2] } }	Map calls the provided sub-expression once for each element in an array, and constructs a new array from the results (e.g. [2, 4]).
\$literal	Treats "\$" as literal instead of evaluating the subexpression.
\$size: { [1, 2, 4] }	Returns the size of the provided array (e.g. 3)

MAKING BACKUPS

The best way to make a backup is to make a copy of the database files while they are in a consistent state (i.e., not in the middle of being read from/to).

1. Use the fsync+lock command. This flushes all in-flight writes to disk and prevents new ones.


```
> db.fsyncLock()
```
2. Copy data files to a new location.
3. Use the unlock command to unlock the database.


```
> db.fsyncUnlock()
```

To restore from this backup, copy the files to the correct server's *dbpath* and start the *mongod*.

If you have a *filesystem* that does *filesystem* snapshots and your journal is on the same volume and you haven't

done anything stripy with RAID, you can take a snapshot without locking. In this case, when you restore, the journal will replay operations to make the data files consistent.

Mongodump is only for backup in special cases. If you decide to use it anyway, don't fsync+lock first.

REPLICA SET MAINTENANCE

KEEPING MEMBERS FROM BEING ELECTED

To permanently stop a member from being elected, change its priority to 0:

```
> var config = rs.config()
> config.members[2].priority = 0
> rs.reconfig(config)
```

To prevent a secondary from being elected temporarily, connect to it and issue the freeze command:

```
> rs.freeze(10*60) // # of seconds to not become primary
```

This can be handy if you don't want to change priorities permanently but need to do maintenance on the primary.

DEMOTING A MEMBER

If a member is currently primary and you don't want it to be, use *stepDown*:

```
> rs.stepDown(10*60) // # of seconds to not try to become primary again
```

STARTING A MEMBER AS A STAND-ALONE SERVER

For maintenance, often it is desirable to start up a secondary and be able to do writes on it (e.g., for building indexes). To accomplish this, you can start up a secondary as a stand-alone *mongod* temporarily.

If the secondary was originally started with the following arguments:

```
$ mongod --dbpath /data/db --replSet setName --port 30000
```

Shut it down cleanly and restart it with:

```
$ mongod --dbpath /data/db --port 30001
```

Note that the *dbpath* does not change, but the port does and the *replSet* option is removed (all other options can remain the same). This *mongod* will come up as a stand-alone server. The rest of the replica set will be looking for a member on port 30000, not 30001, so it will just appear to be "down" to the rest of the set.

When you are finished with maintenance, restart the server with the original arguments.

USER MANAGEMENT

CHECK CURRENT USER PRIVILEGES

```
> db.runCommand(
... {
...   usersInfo:"manager",
...   showPrivileges:true
... }
... )
```

CREATE A SUPERADMIN

```
> use admin
switched to db admin
> db.createUser(
... {
...   user: "superAdmin",
...   pwd: "sa",
...   roles:
...   [
...     { role: "userAdminAnyDatabase",
...       db: "admin"
...     }
...   ]
... }
... )
```

CREATE AN ADMINISTRATOR FOR A GIVEN DATABASE

```
> use sensors
switched to db sensors
> db.createUser(
... {
...   user: "sensorsUserAdmin",
...   pwd: "password",
...   roles:
...   [
...     {
...       role: "userAdmin",
...       db: "sensors"
...     }
...   ]
... }
... )
```

VIEW USER ROLES

```
> use sensors
switched to db sensors
> db.getUser("sensorsUserAdmin")
{
  "_id" : "sensors.sensorsUserAdmin",
  "user" : "sensorsUserAdmin",
  "db" : "sensors",
  "roles" : [
    {
      "role" : "userAdmin",
      "db" : "sensors"
    }
  ]
}
```

SHOW ROLE PRIVILEGES

```
> db.getRole("userAdmin", { showPrivileges: true } )
```

GRANT A ROLE

```
> db.grantRolesToUser(
...   "sensorsUserAdmin",
...   [
...     { role: "read", db: "admin" }
...   ]
... )
```

REVOKE A ROLE

```
> db.grantRolesToUser(
...   "sensorsUserAdmin",
...   [
...     { role: "read", db: "admin" }
...   ]
... )
```

MONGODB RESTRICTIONS

1. The maximum document size is 16 megabytes.
2. Namespaces must be shorter than 123 bytes.
3. Each namespace file must be no larger than 2047 megabytes.
4. The index entry total size must be less than 1024 bytes.
5. A collection can have up to 64 indexes.
6. The index name (namespace included) cannot be longer than 125 chars.
7. A replica set can have at most 12 members.
8. A shard key can have at most 512 bytes.
9. A shard key is always immutable.
10. MongoDB non-indexed field sort will return results only if this operation doesn't use more than 32 megabytes of memory.
11. Aggregation pipeline stages are limited to 100 megabytes of RAM. When the limit is exceeded, an error is thrown. "allowDiskUse" option allows aggregation pipeline stages to use temporary files for processing.
12. A bulk operation is limited to 1000 operations.

13. A database name is case-sensitive and may have up to 64 characters.
14. Collections names cannot contain: \$, null or start with the "system." prefix.

Field names cannot contain: \$, null or . (dot)

ADDITIONAL RESOURCES

- Download MongoDB at <http://www.mongodb.org/downloads>
- Documentation is available at <http://docs.mongodb.org>
- Download TokuMX distribution at http://www.tokutek.com/tokumx_downloads/
- See the roadmap and file bugs and request features at <http://jira.mongodb.org>
- Ask questions on the mailing list: <http://groups.google.com/group/mongodb-user>

ABOUT THE AUTHOR



Vlad Mihalcea is a software architect, passionate about concurrency and data reliability. He's the creator of FlexyPool (<https://github.com/vladmihalcea/flexy-pool>), a reactive connection pooling utility. He's been involved in promoting both SQL and NoSQL solutions and he blogs at: <http://vladmihalcea.com>

RECOMMENDED BOOK



How does MongoDB help you manage a huMONGOus amount of data collected through your web application? With this authoritative introduction, you'll learn the many advantages of using documentoriented databases, and discover why MongoDB is a reliable, high-performance system that allows for almost infinite horizontal scalability.

BUY NOW



BROWSE OUR COLLECTION OF 250+ FREE RESOURCES, INCLUDING:

- RESEARCH GUIDES:** Unbiased insight from leading tech experts
- REFCARDZ:** Library of 200+ reference cards covering the latest tech topics
- COMMUNITIES:** Share links, author articles, and engage with other tech experts

JOIN NOW



DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

Copyright © 2014 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513
888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

