

# Building your first application with MongoDB: Creating a REST API using the MEAN Stack - Part 1

Register for an online course

(<https://university.mongodb.com/?jmp=hero>)

< View all blog posts (/blog)

<https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1?jmp=hero>

Non-Intro to MongoDB (/blog?author=555f57de41636850f3050500)

April 14, 2015

Building your first application with MongoDB: Creating a REST API using the MEAN Stack - Part 1

Introduction

In this 2-part blog series (<https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-2>), you will learn how to use MongoDB, Mongoose Object Data Mapping (ODM) with Express.js and Node.js. These technologies use a uniform language - JavaScript - providing performance gains in the software and productivity gains for developers.

In this first part, we will describe the basic mechanics of our application and undertake data modeling. In the second part (<https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-2>), we will create tests that validate the behavior of our application and then describe how to set-up and run the application.

No prior experience with these technologies is assumed and developers of all skill levels should benefit from this blog series. So, if you have no previous experience using MongoDB, JavaScript or building a REST API, don't worry - we will cover these topics with enough detail to get you past the simplistic examples one tends to find online, including authentication, structuring code in multiple files, and writing test cases.

Let's begin by defining the MEAN stack.

(/search)

## What is the MEAN stack?

(<https://www.mongodb.com/contact?jmp=nav>)

The MEAN stack can be summarized as follows:



- M = MongoDB/Mongoose.js: the popular database, and an elegant ODM for node.js. (<https://www.mongodb.com/lp/download/mongodb-enterprise?jmp=nav>)
- E = Express.js: a lightweight web application framework.
- A = Angular.js: a robust framework for creating HTML5 and JavaScript-rich web applications.
- N = Node.js: a server-side JavaScript interpreter.

The MEAN stack is a modern replacement for the LAMP (Linux, Apache, MySQL, PHP/Python) stack that became the popular way for building web applications in the late 1990s.

In our application, we won't be using Angular.js, as we are not building an HTML user interface. Instead, we are building a REST API which has no user interface, but could instead serve as the basis for any kind of interface, such as a website, an Android application, or an iOS application. You might say we are building our REST API on the ME(a)N stack, but we have no idea how to pronounce that!

## What is a REST API?

REST stands for Representational State Transfer. It is a lighter weight alternative to SOAP and WSDL XML-based API protocols.

REST uses a client-server model, where the server is an HTTP server and the client sends HTTP verbs (GET, POST, PUT, DELETE), along with a URL and variable parameters that are URL-encoded. The URL describes the object to act upon and the server replies with a result code and valid JavaScript Object Notation (JSON).

Because the server replies with JSON, it makes the MEAN stack particularly well suited for our application, as all the components are in JavaScript and MongoDB interacts well with JSON. We will see some JSON examples later, when we start defining our Data Models.

The CRUD acronym is often used to describe database operations. CRUD stands for CREATE, READ, UPDATE, and DELETE. These database operations map very nicely to the HTTP verbs, as follows:

- POST: A client wants to insert or create an object.
- GET: A client wants to read an object.
- PUT: A client wants to update an object.
- DELETE: A client wants to delete an object.

These operations will become clear later when define our API.

Some of the common HTTP result codes that are often used inside REST APIs are as follows:

- 200 - "OK".
- 201 - "Created" (Used with POST).
- 400 - "Bad Request" (Perhaps missing required parameters).
- 401 - "Unauthorized" (Missing authentication parameters).
- 403 - "Forbidden" (You were authenticated but lacking required privileges).
- 404 - "Not Found".

A complete description can be found in the RFC document, listed in the resources section at the end of this blog. We will use these result codes in our application and you will see some examples shortly.

## Why Are We Starting with a REST API?

Developing a REST API enables us to create a foundation upon which we can build all other applications. As previously mentioned, these applications may be web-based or designed for specific platforms, such as Android or iOS.

Today, there are also many companies that are building applications that do not use an HTTP or web interface, such as Uber, WhatsApp, Postmates, and Wash.io. A REST API also makes it easy to implement other interfaces or applications over time, turning the initial project from a single application into a powerful platform.

## Creating our REST API

The application that we will be building will be an RSS Aggregator, similar to Google Reader. Our application will have two main components:

1. The REST API
2. Feed Grabber (similar to Google Reader)

In this blog series we will focus on building the REST API, and we will not cover the intricacies of RSS feeds. However, code for Feed Grabber is available in a github repository, listed in the resources section of this blog.

Let's now describe the process we will follow in building our API. We will begin by defining the data model for the following requirements:

- Store user information in user accounts
- Track RSS feeds that need to be monitored
- Pull feed entries into the database
- Track user feed subscriptions

- Track which feed entry a user has already read

Users will need to be able to do the following:

- Create an account
- Subscribe/unsubscribe to feeds
- Read feed entries
- Mark feeds/entries as read or unread

## Modeling Our Data

An in-depth discussion on data modeling in MongoDB is beyond the scope of this article, so see the references section for good resources on this topic.

We will need 4 collections to manage this information:

- Feed collection
- Feed entry collection
- User collection
- User-feed-entry mapping collection

Let's take a closer look at each of these collections.

## Feed Collection

Lets now look at some code. To model a feed collection, we can use the following JSON document:

```
1 {
2   "_id": ObjectId("523b1153a2aa6a3233a913f8"),
3   "requiresAuthentication": false,
4   "modifiedDate": ISODate("2014-08-29T17:40:22Z"),
5   "permanentlyRemoved": false,
6   "feedURL": "http://feeds.feedburner.com/eater/nyc",
7   "title": "Eater NY",
8   ...
9 }
```

```
8      "bozoBitSet": false,  
9      "enabled": true,  
10     "etag": "4bL78iLSZud2iXd/vd10mYC32BE",  
11     "link": "http://ny.eater.com/",  
12     "permanentRedirectURL": null,  
13     "description": "The New York City Restaurant, Bar, and Nightlife Blog"  
14 }
```

ib.com/danared/67b10475b419470b0d50/raw/6d47d0a3d81e0575b71d84f9213f5edbde857072/gistfile1.txt)  
gistfile1.txt (https://gist.github.com/danared/67b10475b419470b0d50#file-gistfile1-txt) hosted with ❤ by  
GitHub (https://github.com)

If you are familiar with relational database technology, then you will know about databases, tables, rows and columns. In MongoDB, there is a mapping to most of these Relational concepts. At the highest level, a MongoDB deployment supports one or more databases. A database contains one or more collections, which are the similar to tables in a relational database. Collections hold documents. Each document in a collection is, at a highest level, similar to a row in a relational table. However, documents do not follow a fixed schema with pre-defined columns of simple values. Instead, each document consists of one or more key-value pairs where the value can be simple (e.g., a date), or more sophisticated (e.g., an array of address objects).

Our JSON document above is an example of one RSS feed for the Eater Blog, which tracks information about restaurants in New York City. We can see that there are a number of different fields but the key ones that our client application may be interested in include the `URL` of the feed and the `feed description`. The description is important so that if we create a mobile application, it would show a nice summary of the feed.

The remaining fields in our JSON document are for internal use. A very important field is `_id`. In MongoDB, every document must have a field called `_id`. If you create a document without this field, at the point where you save the document, MongoDB will create it for you. In MongoDB, this field is a primary key and MongoDB will guarantee that within a collection, this value is unique.

## Feed Entry Collection

After feeds, we want to track feed entries. Here is an example of a document in the feed entry collection:

```

1  {
2    "_id": ObjectId("523b1153a2aa6a3233a91412"),
3    "description": "Buzzfeed asked a bunch of people...",
4    "title": "Cronut Mania: Buzzfeed asked a bunch of people...",
5    "summary": "Buzzfeed asked a bunch of people that were...",
6    "content": [{
7      "base": "http://ny.eater.com/",
8      "type": "text/html",
9      "value": "LOTS OF HTML HERE ",
10     "language": "en"
11   }],
12   "entryID": "tag:ny.eater.com,2013://4.560508",
13   "publishedDate": ISODate("2013-09-17T20:45:20Z"),
14   "link": "http://ny.eater.com/archives/2013/09/cronut_mania_41.php",
15   "feedID": ObjectId("523b1153a2aa6a3233a913f8")
16 }

```

<https://gist.github.com/danared/3cf475bac219cb3b69f2/raw/936de35593a03e735f59508a56159582f51d4598/gistfile1.txt>  
[gistfile1.txt \(https://gist.github.com/danared/3cf475bac219cb3b69f2#file-gistfile1-txt\)](https://gist.github.com/danared/3cf475bac219cb3b69f2#file-gistfile1-txt) hosted with ♥ by  
 GitHub (<https://github.com>)

Again, we can see that there is a `_id` field. There are also some other fields, such as `description`, `title` and `summary`. For the content field, note that we are using an array, and the array is also storing a document. MongoDB allows us to store sub-documents in this way and this can be very useful in some situations, where we want to hold all information together. The `entryID` field uses the tag format to avoid duplicate feed entries. Notice also the `feedID` field that is of type `ObjectId` - the value is the `_id` of the Eater Blog document, described earlier. This provides a referential model, similar to a foreign key in a relational database. So, if we were interested to see the feed document associated with this `ObjectId`, we could take the value `523b1153a2aa6a3233a913f8` and query the feed collection on `_id`, and it would return the Eater Blog document.

## User Collection

Here is the document we could use to keep track of users:

```

1  {
2    "_id" : ObjectId("54ad6c3ae764de42070b27b1"),
3    "active" : true,
4    "email" : "testuser1@example.com",
5    "firstName" : "Test",
6    "lastName" : "User1"

```

```

6      lastname : user1 ,
7      "sp_api_key_id" : "6YQB0A8VXM0X8RVDPPPLRHBI7J",
8      "sp_api_key_secret" : "veBw/YFx56Dl0bbiVEpvbjF",
9      "lastLogin" : ISODate("2015-01-07T17:26:18.996Z"),
10     "created" : ISODate("2015-01-07T17:26:18.995Z"),
11     "subs" : [ ObjectId("523b1153a2aa6a3233a913f8"),
12               ObjectId("54b563c3a50a190b50f4d63b") ],
13   }

```

<https://gist.github.com/danared/5d67ef16a47b7a650d36/raw/f35299ae26cad6bdb992a8c2297627ce2f5d76/gistfile1.txt>  
 gistfile1.txt (<https://gist.github.com/danared/5d67ef16a47b7a650d36#file-gistfile1-txt>) hosted with ♥ by  
 GitHub (<https://github.com>)

A user has an email address, first name and last name. There is also an sp\_api\_key\_id and sp\_api\_key\_secret - we will use these later with Stormpath (<https://stormpath.com/>), a user management API. The last field, called subs, is a subscription array. The subs field tells us which feeds this user is subscribed-to.

### User-Feed-Entry Mapping Collection

The last collection allows us to map users to feeds and to track which feeds have been read.

```

1  {
2      "_id" : ObjectId("523b2fcc054b1b8c579bdb82"),
3      "read" : true,
4      "user_id" : ObjectId("54ad6c3ae764de42070b27b1"),
5      "feed_entry_id" : ObjectId("523b1153a2aa6a3233a91412"),
6      "feed_id" : ObjectId("523b1153a2aa6a3233a913f8")
7  }

```

<https://gist.github.com/danared/e3bd05a074e3608cb862/raw/673e09645b2a031e87da2d203fe5111cf6ca24d3/gistfile1.txt>  
 gistfile1.txt (<https://gist.github.com/danared/e3bd05a074e3608cb862#file-gistfile1-txt>) hosted with ♥ by  
 GitHub (<https://github.com>)

We use a Boolean (true/false) to mark the feed as read or unread.

## Functional Requirements for the REST API

As previously mentioned, users need to be able to do the following:

- Create an account.



- Subscribe/unsubscribe to feeds.
- Read feed entries.
- Mark feeds/entries as read or unread.

Additionally, a user should be able to reset their password.

The following table shows how these operations can be mapped to HTTP routes and verbs.

Route	Verb	Description	Variables
/user/enroll	POST	Register a new user	firstName lastName email password
/user/resetPassword	PUT	Password Reset	email
/feeds	GET	Get feed subscriptions for each user with description and unread count	
/feeds/subscribe	PUT	Subscribe to a new feed	feedURL
/feeds/entries	GET	Get all entries for feeds the user is subscribed to	
/feeds/<feedid>/entries	GET	Get all entries for a specific feed	
/feeds/<feedid>	PUT	Mark all entries for a specific feed as read or unread	read = <true   false>
/feeds/<feedid>/entries/<entryid>	PUT	Mark a specific entry as either read or unread	read = <true   false>
/feeds/<feedid>	DELETE	Unsubscribe from this particular feed	

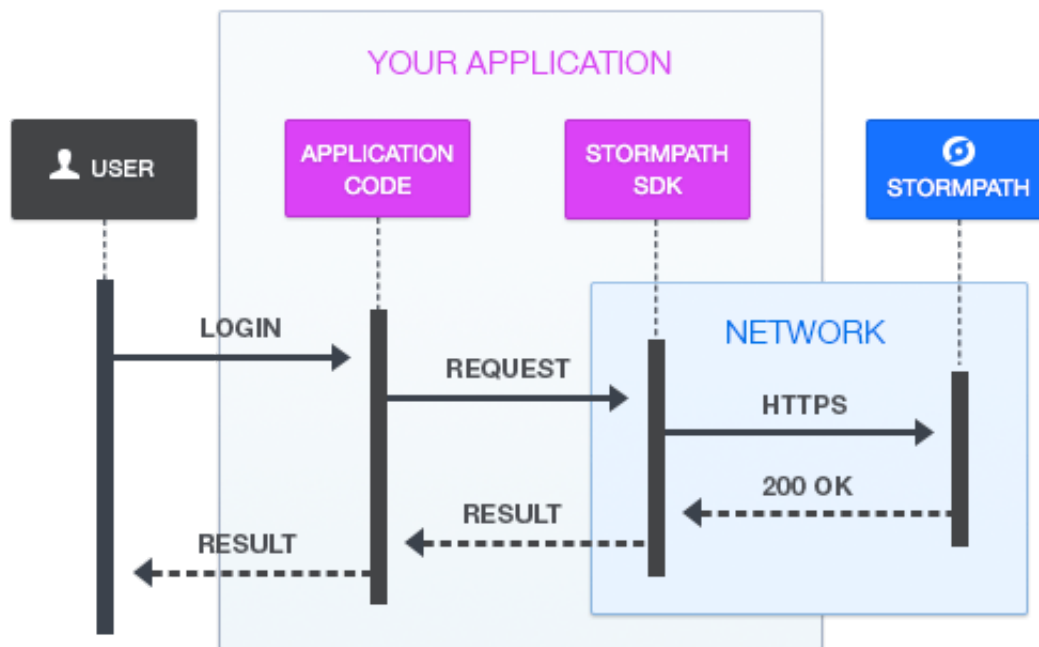
In a production environment, the use of secure HTTP (HTTPS) would be the standard approach when sending sensitive details, such as passwords.

## Real World Authentication with Stormpath

In robust real-world applications it is important to provide user authentication. We need a secure approach to manage users, passwords, and password resets.

There are a number of ways we could authenticate users for our application. One possibility is to use Node.js with the Passport Plugin, which could be useful if we wanted to authenticate with social media accounts, such as Facebook or Twitter. However, another possibility is to use Stormpath. Stormpath provides User Management as a Service and supports authentication and authorization through API keys. Basically, Stormpath maintains a database of user details and passwords and a client application REST API would call the Stormpath REST API to perform user authentication.

The following diagram shows the flow of requests and responses using Stormpath.



In detail, Stormpath will provide a secret key for each “Application” that is defined with their service. For example, we could define an application as “Reader Production” or “Reader Test”. This could be very useful when we are still developing and testing our application, as we may be frequently adding and deleting test users. Stormpath will also provide an `API Key Properties` file.

Stormpath also allows us to define password strength requirements for each application, such as:

- Must have  $\geq 8$  characters.

- Must include lowercase and uppercase.
- Must include a number.
- Must include a non-alphabetic character

Stormpath keeps track of all of our users and assigns them API keys, which we can use for our REST API authentication. This greatly simplifies the task of building our application, as we don't have to focus on writing code for authenticating users.

## Node.js

Node.js is a runtime environment for server-side and network applications. Node.js uses JavaScript and it is available for many different platforms, such as Linux, Microsoft Windows and Apple OS X.

Node.js applications are built using many library modules and there is a very rich ecosystem of libraries available, some of which we will use to build our application.

To start using Node.js, we need to define a `package.json` file describing our application and all of its library dependencies.

The Node.js Package Manager installs copies of the libraries in a subdirectory, called `node_modules/`, in the application directory. This has benefits, as it isolates the library versions for each application and so avoids code compatibility problems if the libraries were to be installed in a standard system location, such as `/usr/lib`, for example.

The command `npm install` will create the `node_modules/` directory, with all of the required libraries.

Here is the JavaScript from our `package.json` file:

```
1  {
2    "name": "reader-api",
3    "main": "server.js",
4    "dependencies": {
5      "express" : "~4.10.0",
6      "stormpath" : "~0.7.5", "express-stormpath" : "~0.5.9",
7      "mongodb" : "~1.4.26", "mongoose" : "~3.8.0",
```

```
8      "body-parser" : "~1.10.0", "method-override" : "~2.3.0",
9      "morgan" : "~1.5.0", "winston" : "~0.8.3", "express-winston" : "~0.2.9",
10     "validator" : "~3.27.0",
11     "path" : "~0.4.9",
12     "errorhandler" : "~1.3.0",
13     "frisby" : "~0.8.3",
14     "jasmine-node" : "~1.14.5",
15     "async" : "~0.9.0"
16   }
17 }
```

ub.com/danared/289354cbe51fbef4216b/raw/cb42f79f9039b2b4be2666d024b4c253d1dd8a84/gistfile1.txt)  
gistfile1.txt (<https://gist.github.com/danared/289354cbe51fbef4216b#file-gistfile1-txt>) hosted with ❤ by  
GitHub (<https://github.com>)

Our application is called **reader-api**. The main file is called `server.js`. Then we have a list of the dependent libraries and their versions. Some of these libraries are designed for parsing the HTTP queries. The test harness we will use is called `frisby`. The `jasmine-node` is used to run frisby scripts.

One library that is particularly important is `async`. If you have never used node.js, it is important to understand that node.js is designed to be asynchronous. So, any function which does blocking input/output (I/O), such as reading from a socket or querying a database, will take a callback function as the last parameter, and then continue with the control flow, only returning to that callback function once the blocking operation has completed. Let's look at the following simple example to demonstrate this.

```
1  function foo() {
2      someAsyncFunction(params, function(err, results) {
3          console.log("one");
4      });
5      console.log("two");
6  }
```

ub.com/danared/60eac629c1b55d88c29d/raw/2418c02f9438fe4a082031ff2be2112fbb5c606a/gistfile1.txt)  
gistfile1.txt (<https://gist.github.com/danared/60eac629c1b55d88c29d#file-gistfile1-txt>) hosted with ❤ by  
GitHub (<https://github.com>)

In the above example, we may think that the output would be:

one  
two

but in fact it might be:

two

one

because the line that prints “one” might happen later, asynchronously, in the callback. We say “might” because if conditions are just right, “one” might print before “two”. This element of uncertainty in asynchronous programming is called non-deterministic execution. For many programming tasks, this is actually desirable and permits high performance, but clearly there are times when we want to execute functions in a particular order. The following example shows how we could use the `async` library to achieve the desired result of printing the numbers in the correct order:

```
1  actionArray = [  
2      function one(cb) {  
3          someAsyncFunction(params, function(err, results) {  
4              if (err) {  
5                  cb(new Error("There was an error"));  
6              }  
7              console.log("one");  
8              cb(null);  
9          });  
10     },  
11     function two(cb) {  
12         console.log("two");  
13         cb(null);  
14     }  
15 ]  
16  
17 async.series(actionArray);  
18
```

<https://gist.github.com/danared/d81d389bbbcc748568fe/raw/e2ec53cadff5aceaf6905c5c681ab3b41edc4e72/gistfile1.txt>  
gistfile1.txt (<https://gist.github.com/danared/d81d389bbbcc748568fe#file-gistfile1-txt>) hosted with ♥ by  
GitHub (<https://github.com>)

In the above code, we are guaranteed that `function two` will only be called after `function one` has completed.

## Wrapping Up Part 1

Now that we have seen the basic mechanics of node.js and async function setup, we are ready to move on. Rather than move into creating the application, we will instead start by creating tests that validate the behavior of the application. This approach is called test-driven development and has two very good features:

- It helps the developer really understand how data and functions are consumed and often exposes subtle needs like the ability to return 2 or more things in an array instead of just one thing.
- By writing tests before building the application, the paradigm becomes “broken / unimplemented until proven tested OK” instead of “assumed to be working until a test fails.” The former is a “safer” way to keep the code healthy.

---

Learn more MongoDB. Register for our free online courses:

Learn MongoDB for free (<https://university.mongodb.com/?jmp=blog>)

---

**Read Part 2 >> (<https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-2>)**

*About the Author - Norberto*

Norberto Leite is Technical Evangelist at MongoDB. Norberto has been working for the last 5 years on large scalable and distributable application environments, both as advisor and engineer. Prior to MongoDB Norberto served as a Big Data Engineer at Telefonica.

Search



---

## Featured

3.0 (/blog/post/whats-new-mongodb-3.0-part-1-95-reduction-operational-overhead-and-security-enhancements)

## Events

(//www.mongodb.com/blog/post/spend-an-evening-with-mongodb)

## Spark

(//www.mongodb.com/blog/post/leaf-in-the-wild-stratio-integrates-apache-spark-and-mongodb-to-unlock-new-customer-insights-for-one-of-worlds-largest-banks)

## Community

(//www.mongodb.com/blog/post/now-accepting-nominations-for-the-2015-william-zola-award-for-community-excellence)

## Security

(//www.mongodb.com/blog/post/july-mongodb-security-best-practices)

## Ops Best Practices

(//www.mongodb.com/blog/post/your-ultimate-guide-to-rolling-upgrades)

## News

(//www.mongodb.com/blog/post/dbta-presents-mongodb-readers-choice-award-for-second-year-in-row)

---

## Receive updates from MongoDB

Business email:

First name:

Last name:

Business phone:

Company:

Job function:

Country:

Subscribe

7 Comments

MongoDB.com Blog

 Login

 Recommend 7  Share

Sort by Newest



Join the discussion...



**mselmany** • 3 months ago

Also look this for advanced mean development step by step: <https://github.com/hwz/chirp>

  • Reply • Share >



**Jag** • 7 months ago

If any of you guys are looking for a tutorial on REST service using pure Node.js (from the scratch), I came across an excellent video here:





1 ^ v • Reply • Share >



**mightyscoo** → Jag • 6 months ago

Interesting, thanks!

^ v • Reply • Share >



**TheMikester** • a year ago

Not really a MEAN stack without Angular, is it? You should also try something like Restify which is made for implementing REST APIs, instead of bastardizing Express.

^ v • Reply • Share >



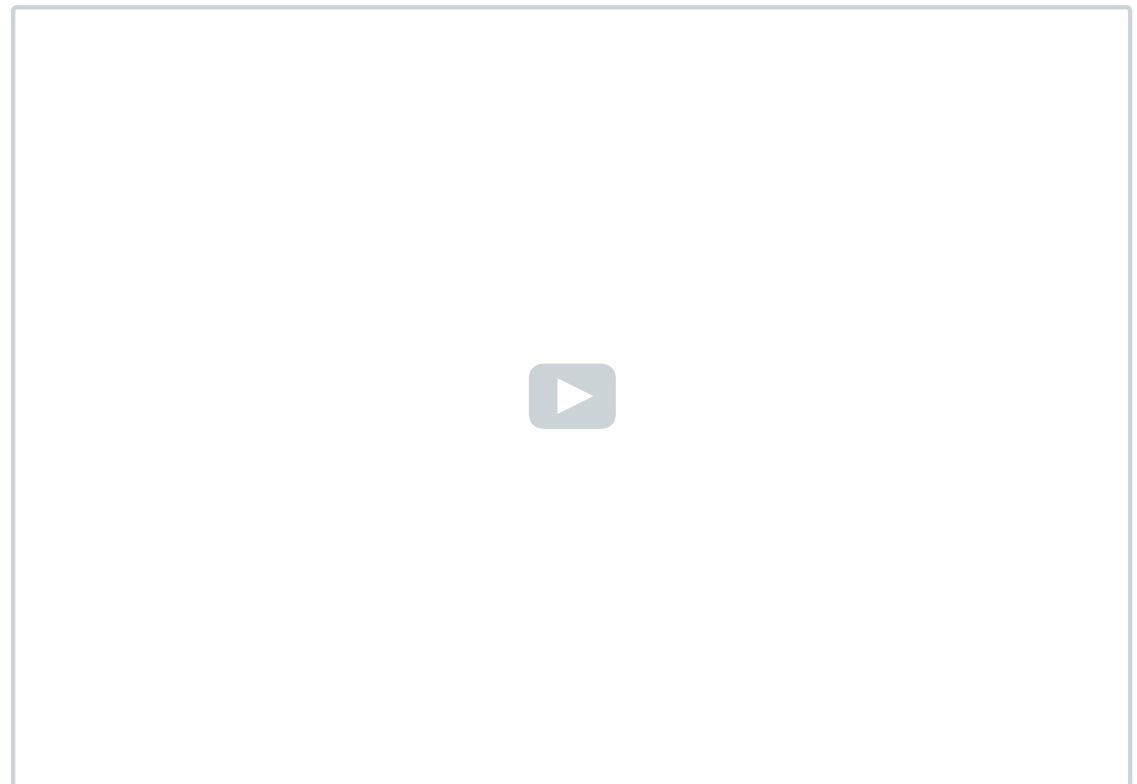
**Qbert1** • a year ago

How about another post that covers user authentication? That's the part I'm really interested in

^ v • Reply • Share >



**WCat** → Qbert1 • 3 months ago



This is fantastic!

^ v • Reply • Share >



**meghangill** → Qbert1 • a year ago

Check out our recent blog post here: <http://www.mongodb.com/blog/po...>

^ v • Reply • Share >

ALSO ON MONGODB.COM BLOG

WHAT'S THIS?

## How We Brought the MongoDB University App to Life

3 comments • 5 months ago

**Andrew Shapton** — Looking forward to the Android app and a bunch of new announcements !

## Announcing MongoDB's Giant of the Month, Attila Toszer of Amadeus

11 comments • 5 months ago

**Sig Narvaez** — Congrats!!

## What's New in MongoDB 3.2, Part 1: Extending Use Cases with New ...

3 comments • 3 months ago

**Douglas Duncan** — Thanks for the great write up of the new 3.2 features Mat! I'm looking forward to getting in and testing ...

## Document Validation – Part 2: Putting it all Together, a Tutorial

7 comments • 5 months ago

**Wilfried Haverkamp** — Probably, visualization with your new tool is the best way to start an investigation. To use ...



Subscribe



Add Disqus to your site Add Disqus Add



Privacy

### About MongoDB

[About MongoDB, Inc \(/company\)](/company)[Careers \(/careers\)](/careers)[Contact Us \(/contact\)](/contact)[Legal Notices \(/legal/legal-notice\)](/legal/legal-notice)[Security Information \(/security\)](/security)

### Learn More


[NoSQL Databases Explained \(/nosql-explained\)](/nosql-explained)[MongoDB Architecture Guide \(/lp/white-paper/architecture-guide\)](/lp/white-paper/architecture-guide)[MongoDB Enterprise Advanced \(/products/mongodb-enterprise-advanced\)](/products/mongodb-enterprise-advanced)[MongoDB Cloud Manager \(/cloud\)](/cloud)[Engineering @ MongoDB \(https://engineering.mongodb.com\)](https://engineering.mongodb.com)

### MongoDB University

[View Course Catalog \(/university.mongodb.com/courses/catalog\)](https://university.mongodb.com/courses/catalog)[View Course Schedule \(/university.mongodb.com/courses/schedule\)](https://university.mongodb.com/courses/schedule)[Public Training \(/university.mongodb.com/training\)](https://university.mongodb.com/training)

Certification ([//university.mongodb.com/exams](http://university.mongodb.com/exams))

## Follow Us

 Github ([//github.com/mongodb](https://github.com/mongodb))

 Twitter ([//twitter.com/MongoDB](https://twitter.com/MongoDB))

 Facebook ([//www.facebook.com/mongodb](https://www.facebook.com/mongodb))

 YouTube  
([//www.youtube.com/user/MongoDB](https://www.youtube.com/user/MongoDB))

**Popular Topics:** Building Modern Apps (<https://www.mongodb.com/scale/building-modern-apps>), Types Of NoSQL Databases (<https://www.mongodb.com/scale/types-of-nosql-databases>), NoSQL Database Gartner Magic Quadrant (<https://www.mongodb.com/scale/nosql-database-gartner-magic-quadrant>), Gartner Operational Database Management Landscape (<https://www.mongodb.com/scale/gartner-operational-database-management-landscape>), Sql Server Data Types (<https://www.mongodb.com/scale/sql-server-data-types>), Big Data Analytics Architecture (<https://www.mongodb.com/scale/big-data-analytics-architecture>), NoSQL Performance Benchmarks (<https://www.mongodb.com/scale/nosql-performance-benchmarks>), Database Software Open Source (<https://www.mongodb.com/scale/database-software-open-source>), Types Of NoSQL Database Management Systems (<https://www.mongodb.com/scale/types-of-nosql-database-management-systems>)

Copyright © 2016 MongoDB, Inc.

Mongo, MongoDB, and the MongoDB leaf logo are registered trademarks of MongoDB, Inc.