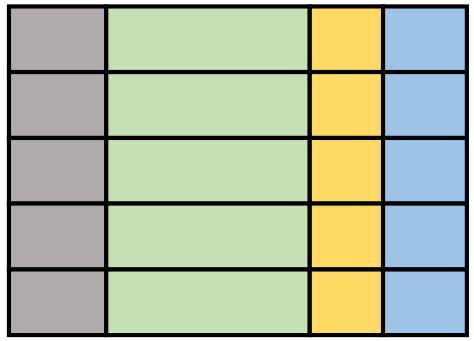
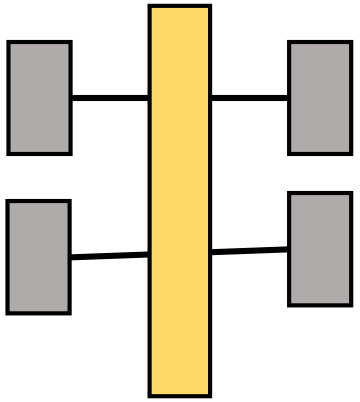
Day 26

Types of Databases



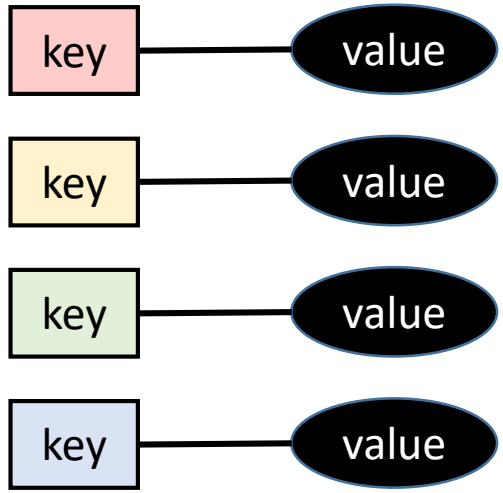
Relational

NoSQL database
- typically used for data warehousing



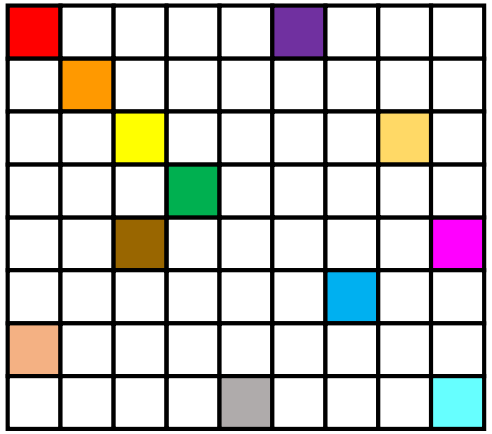
Analytical (OLAP)

eg. Redis
- typically memory based (data not stored in harddisk)
- usually only storing important data



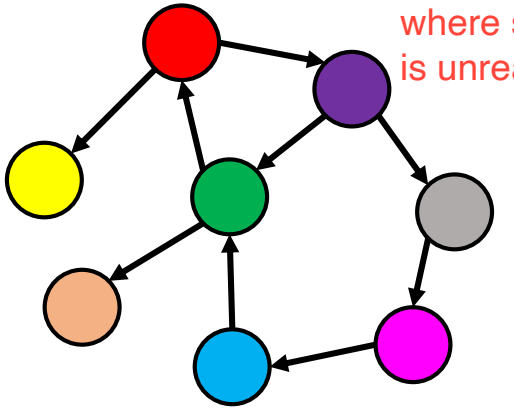
Key-Value

inverts rows and columns
headers are in rows, records are in columns

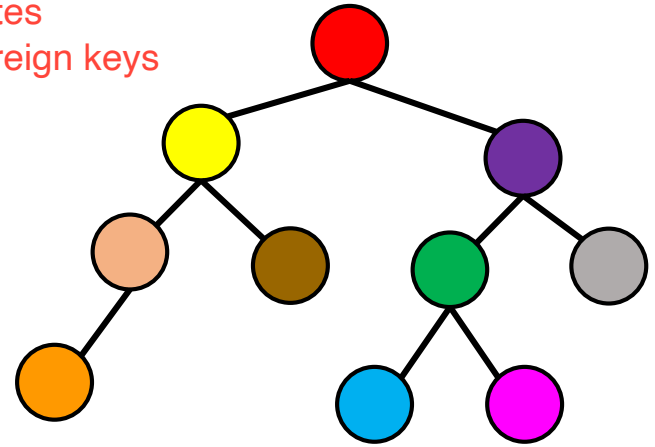


Column

for databases with a lot of rs
e.g. social media sites
where setting up foreign keys
is unrealistic



Graph



Document

From https://www.slideshare.net/Couchbase/webinar-making-sense-of-nosql-applying-nonrelational-databases-to-business-needs

Examples of NoSQL Database

Key-Value

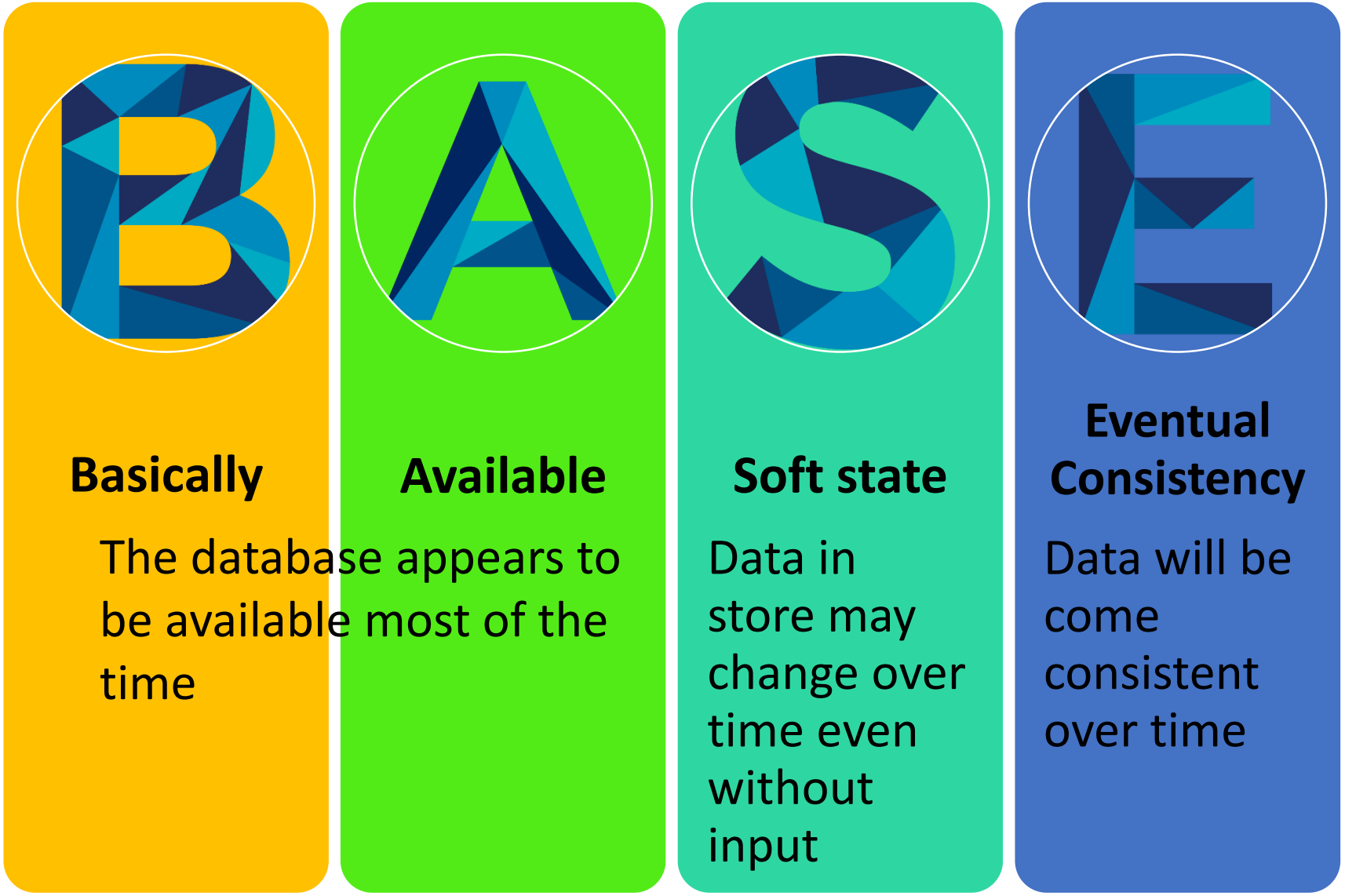
Document

Graph

Column

Analytical

BASE Properties



Key-Value vs Document Database

Key-Value

- Uses keys to access the value
- Values are blobs - can store anything
  - Image, video, PDF, Word document
- Values are opaque - cannot be queried

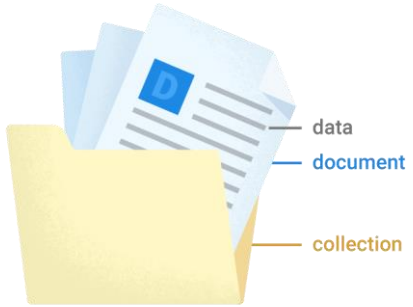
Document

- Data are typically stored as JSON
- Data can be nested and hierarchical
- Any item in the document can be queried

What is MongoDB?

- Document database
  - A document is a JSON object
- A record is a JSON object
  - Most RDBMS allows JSON data type as part of a record - a column
- Use cases
  - Big data
  - Content management
  - Complex data
  - Mobile apps

Terminology



RDBMS (MySQL)	Document (MongoDB)
Database	Database
Table	Collection
Record / Row	Document
Column	Field / Attribute
Index	Index
Joins	Linking and Embedding

Image from https://firebase.google.com/docs/firestore/images/structure-data.png

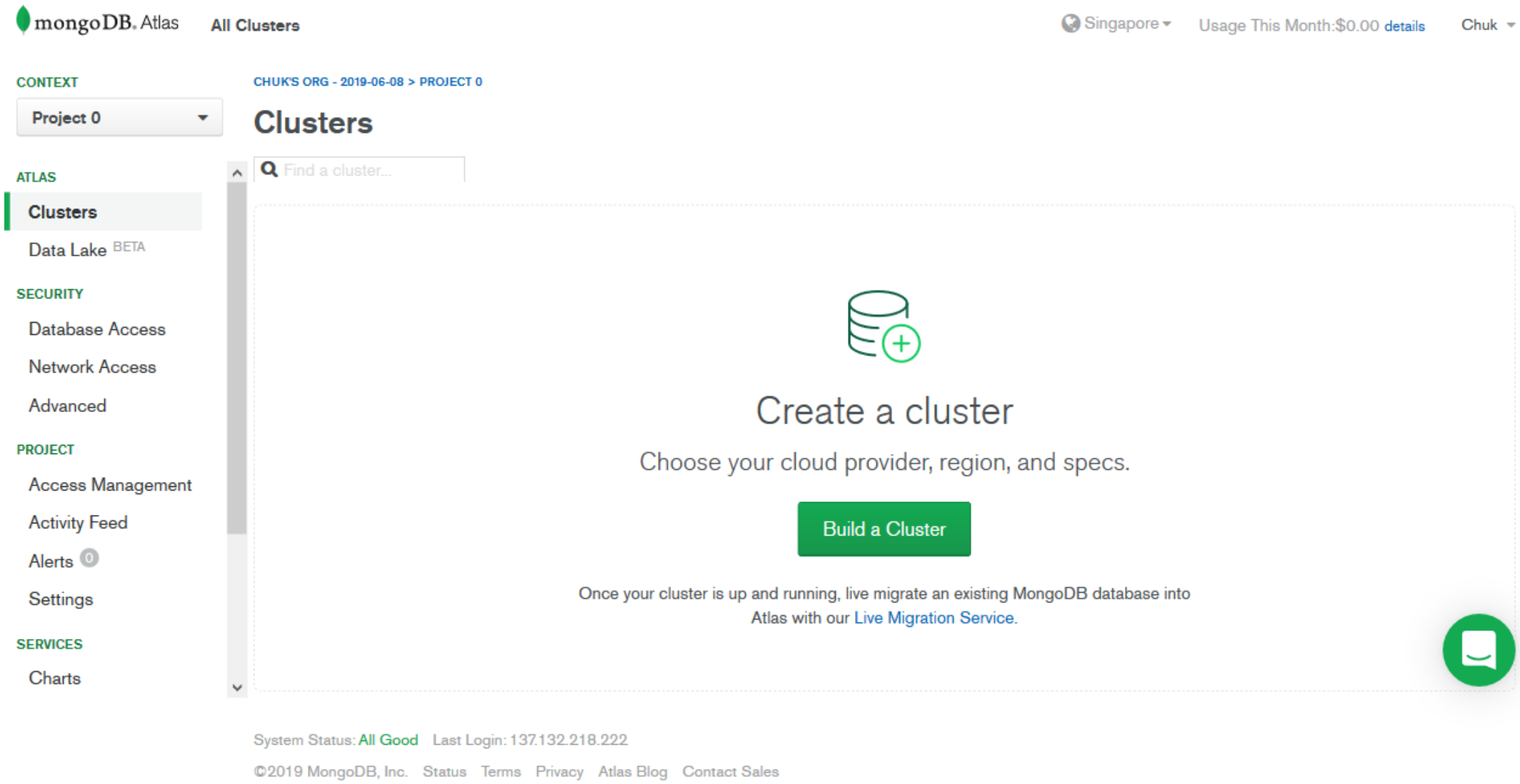
An Example MongoDB Document

```
{
    _id: ObjectId("54759eb3c090d83494e2d804",
    title: "MongoDB: The Definitive Guide",
    authors: [
        { _id: "kchodorow", name: "Kristina Chodorow" },
        { _id: "mdirold", name: "Mike Dirolf" }
    ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    thumbnail: BinData(0,"AREhMQ==")
    publisher: {
        name: "O'Reilly Media",
        founded: 1980,
        locations: ["CA", "NY" ]
    }
}
```

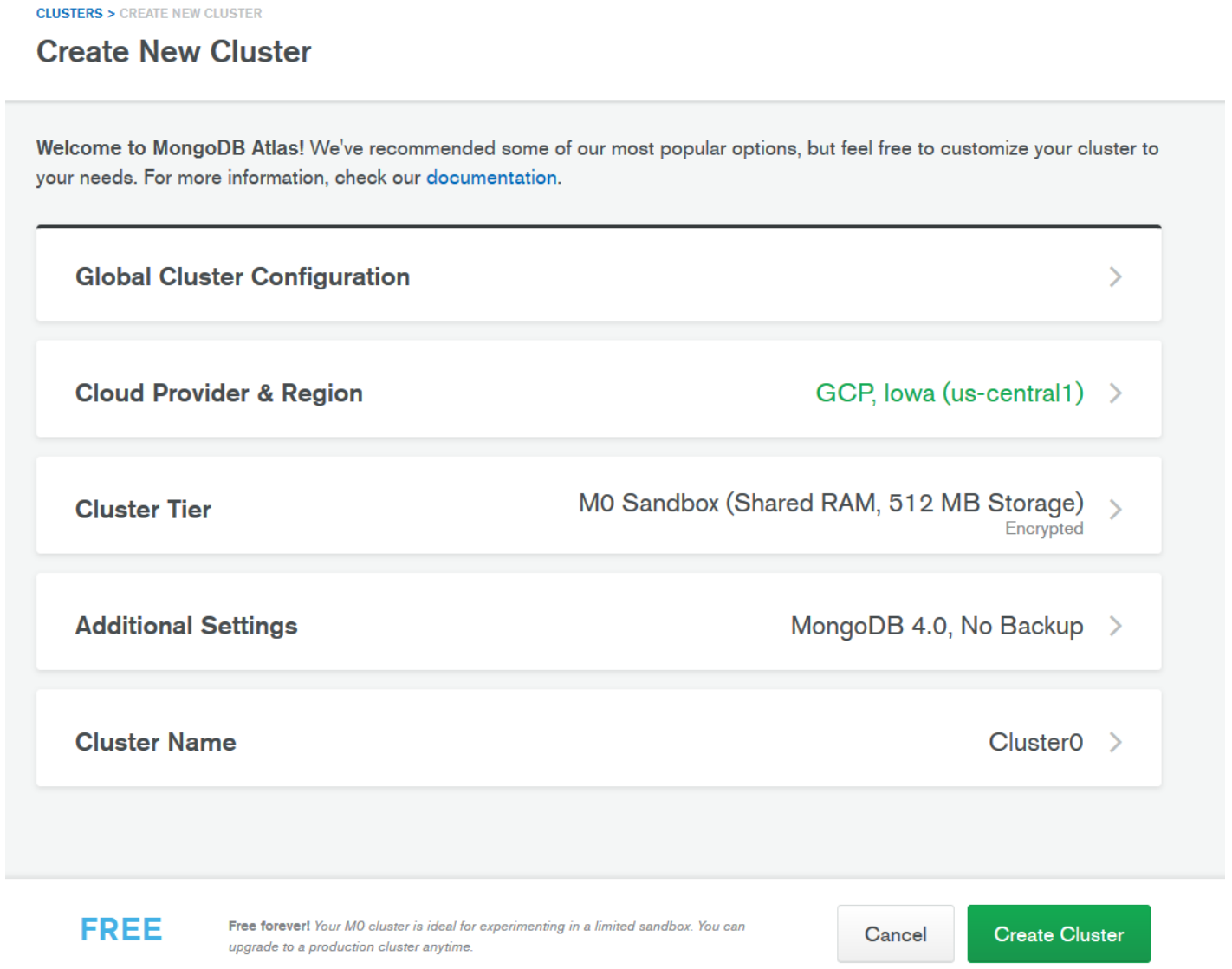
Primary key

MongoDB data type - BSON

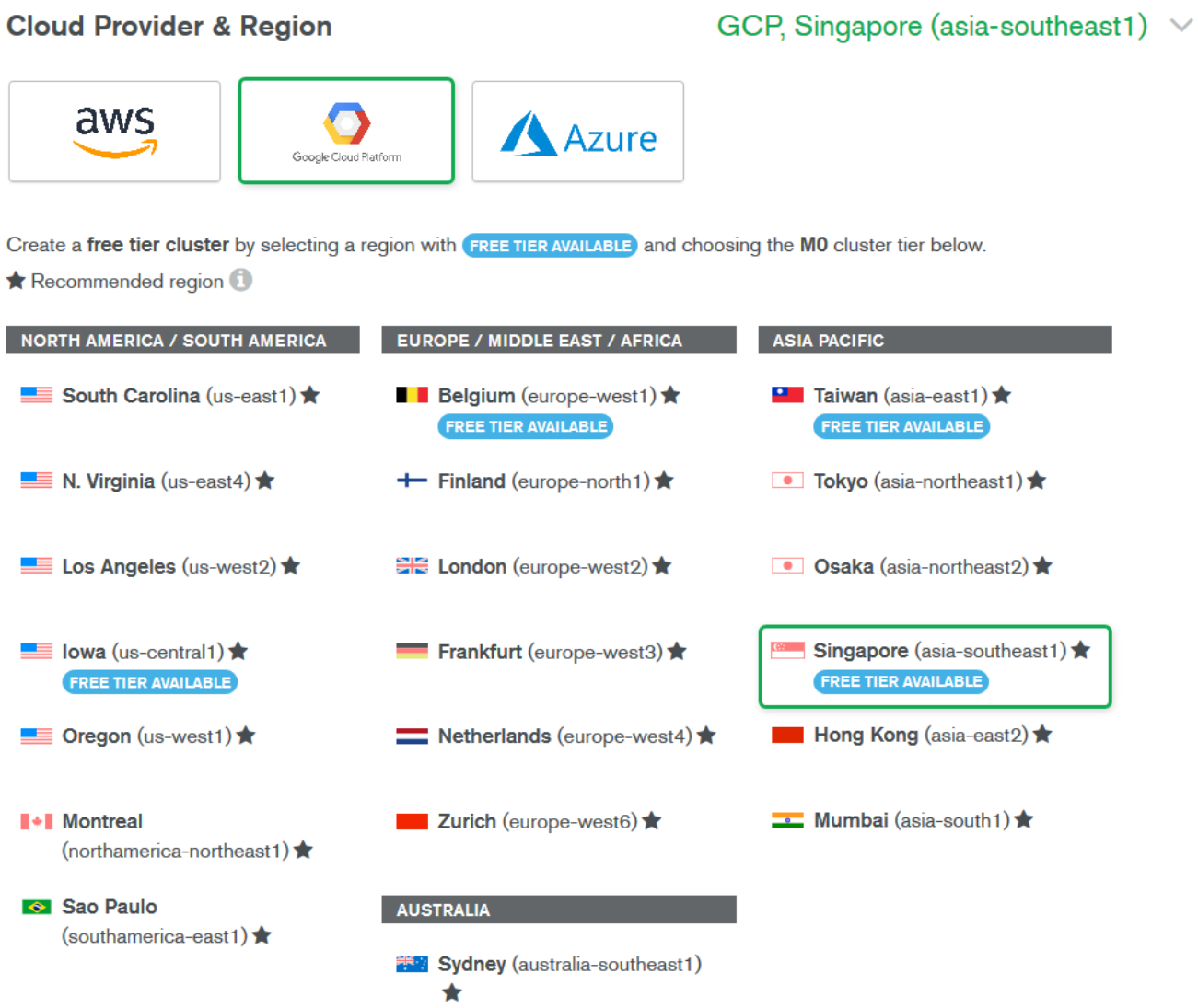
Create a Cluster



Cluster Configurations



Cluster Configuration - Provider and Region



Cloud Configuration - Cluster Tier

Cluster Tier — M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted

Base hourly rate is for a MongoDB replica set with **3 data bearing servers**.

Shared Clusters for development environments and low-traffic applications

Tier	RAM	Storage	vCPU	Base Price
M0 Sandbox	Shared	512 MB	Shared	Free forever
M0 clusters are best for getting started, and are not suitable for production environments.				
100 max connections | Low network performance | 100 max databases | 500 max collections				
M2	Shared	2 GB	Shared	$9 / MONTH
M5	Shared	5 GB	Shared	$25 / MONTH

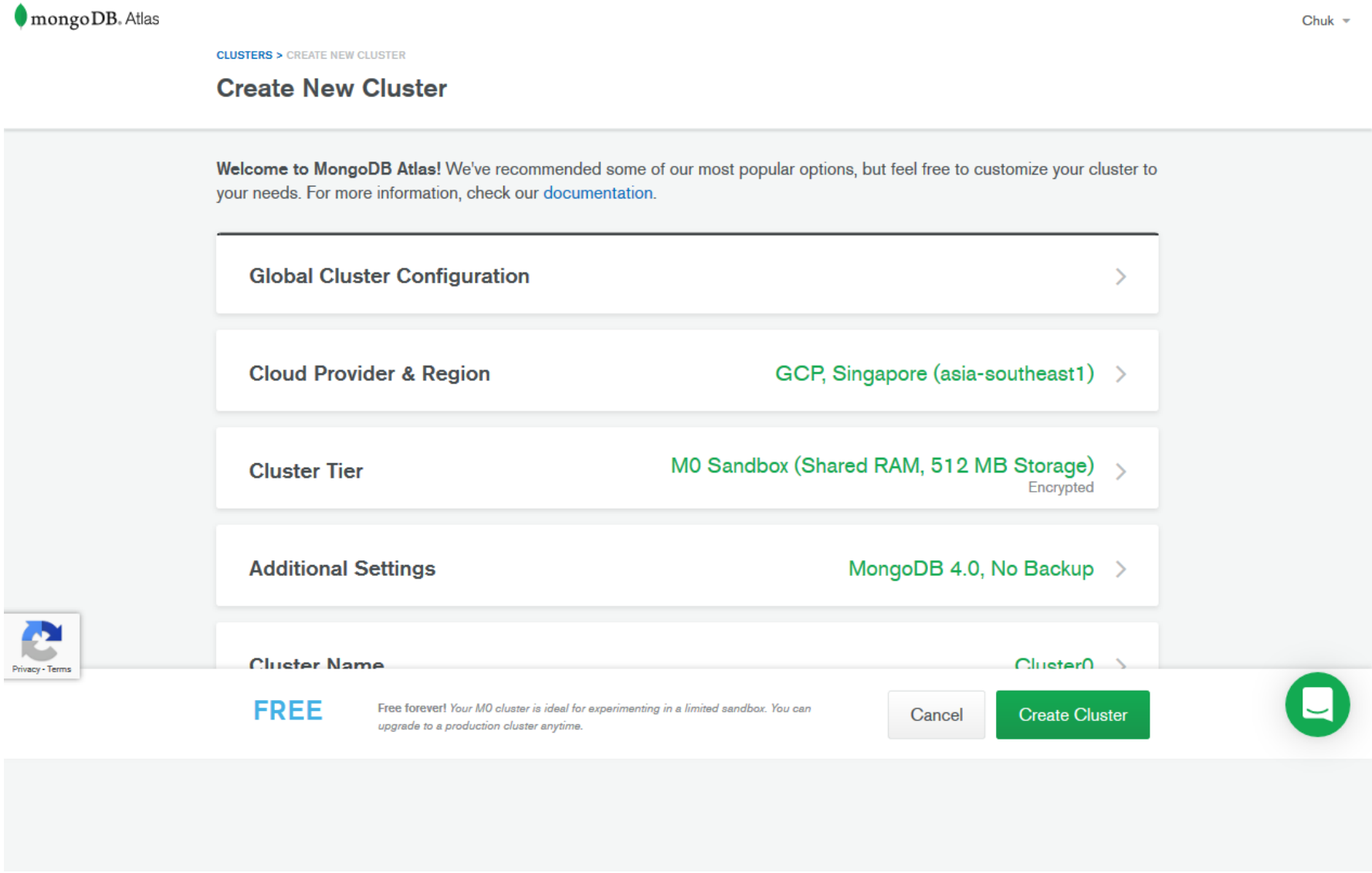
Dedicated Clusters for development environments and low-traffic applications

Tier	RAM	Storage	vCPU	Base Price
M10	1.7 GB	10 GB	0.5 vCPUs	from **$0.10**/hr
M20	3.75 GB	20 GB	1 vCPU	from **$0.23**/hr

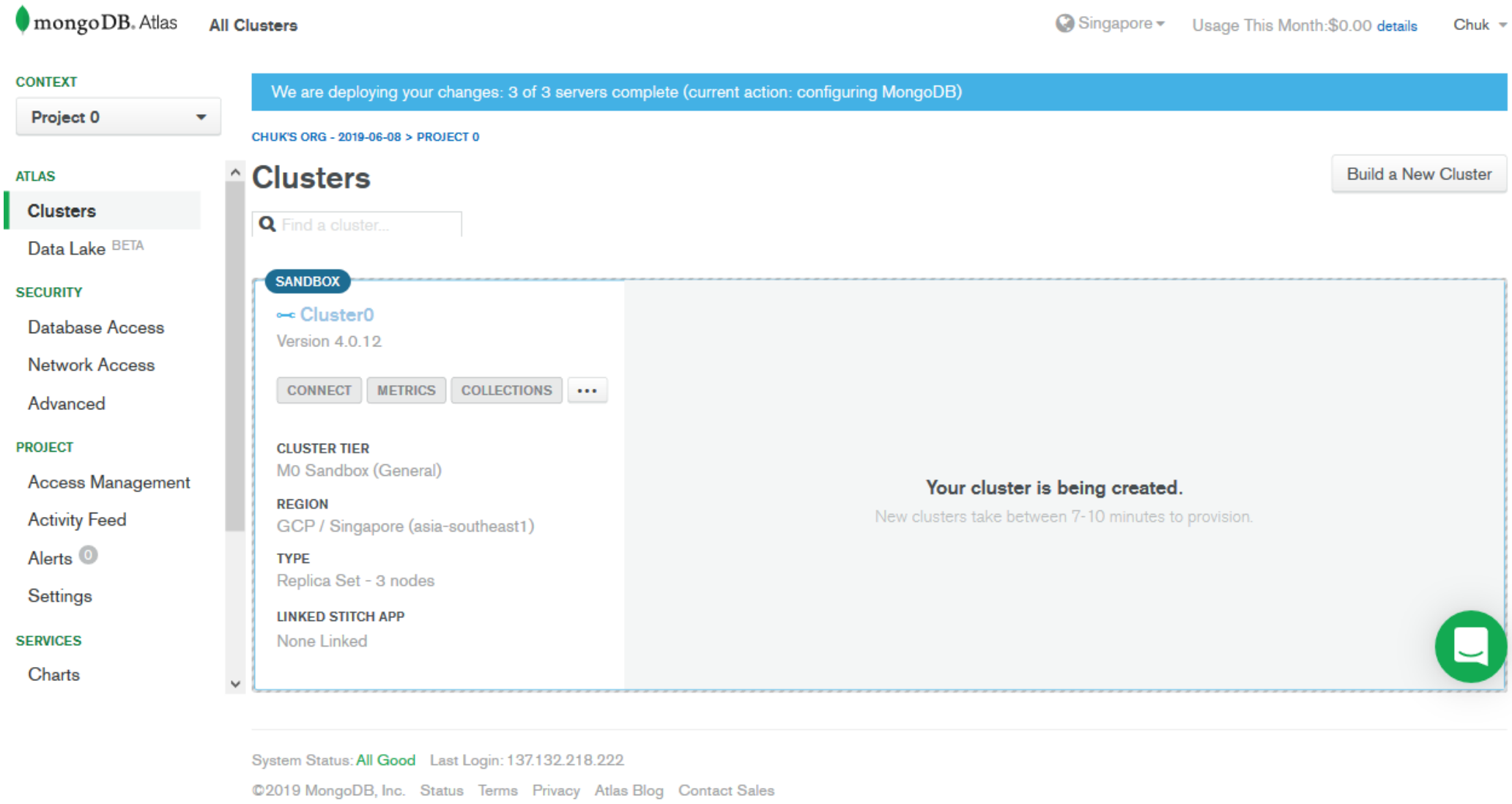
Dedicated Clusters for high-traffic applications and large datasets

Tier	RAM	Storage	vCPU	Base Price

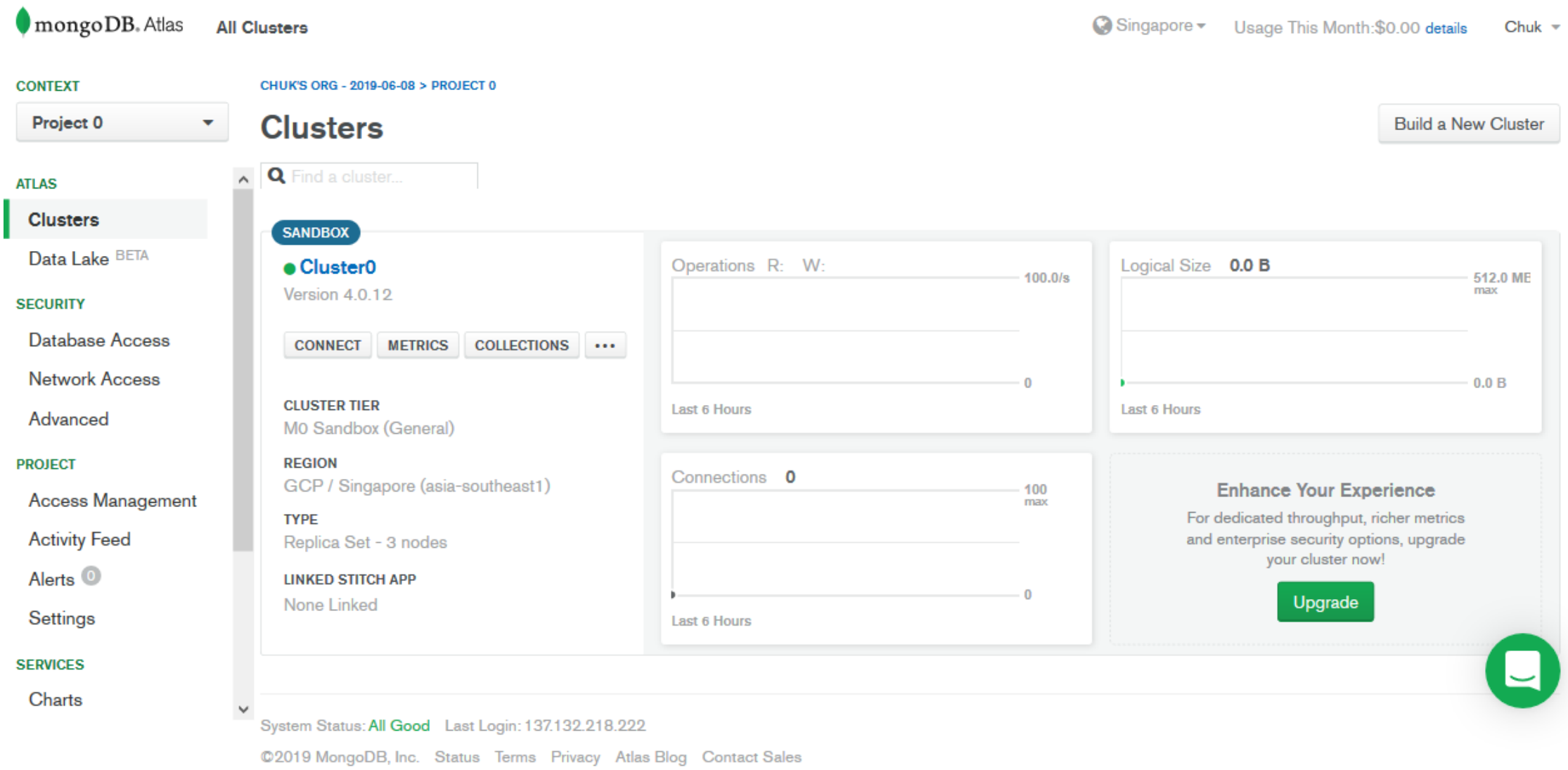
Create Cluster



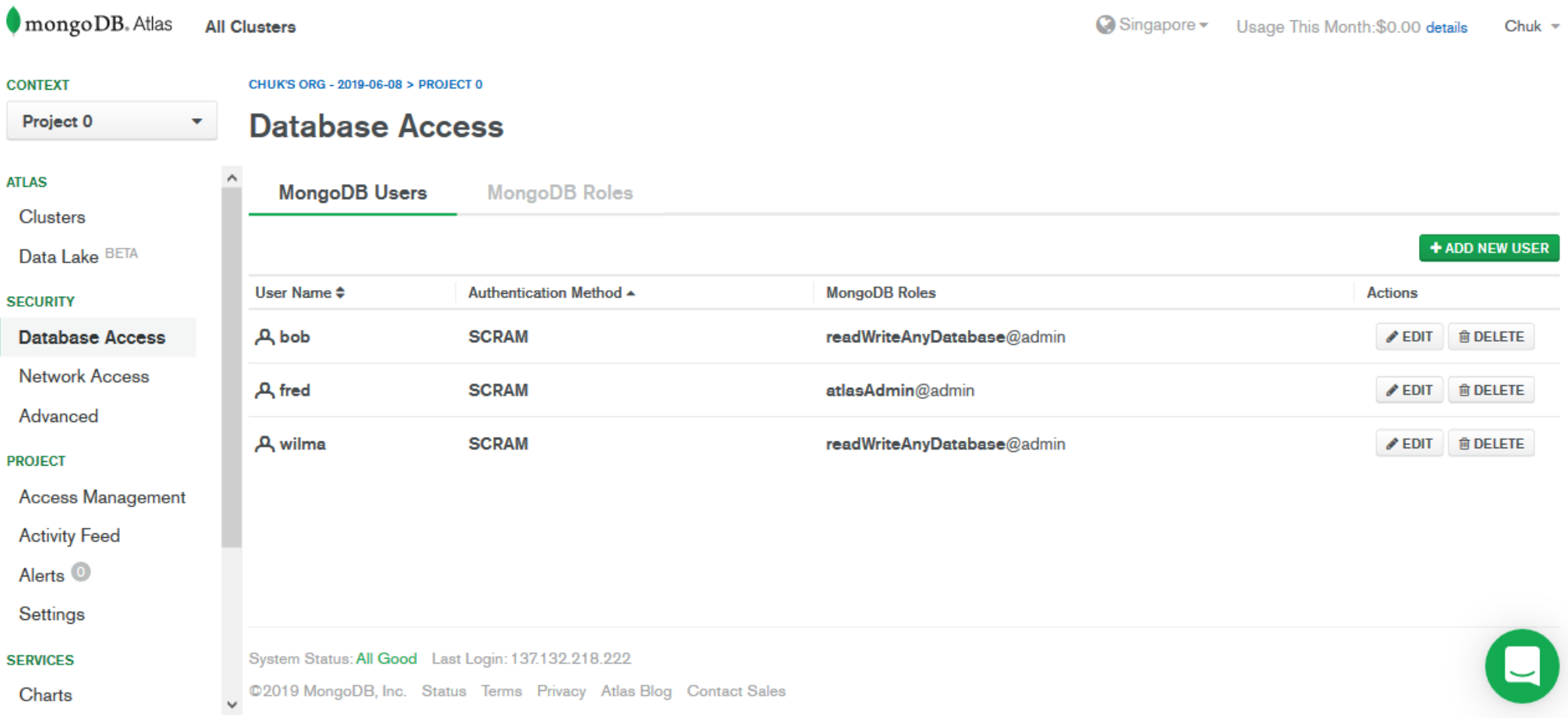
Creating Cluster



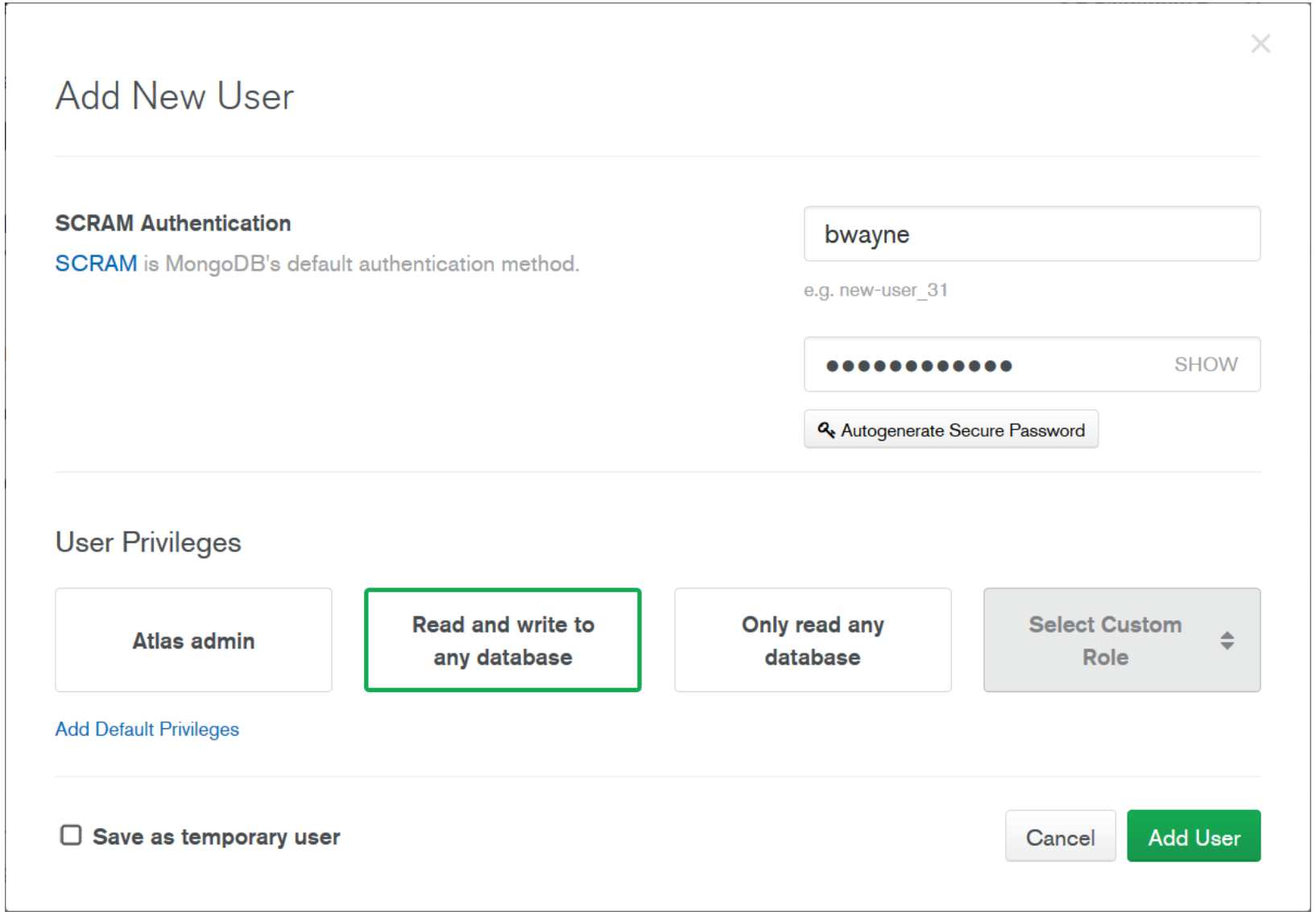
Cluster Ready



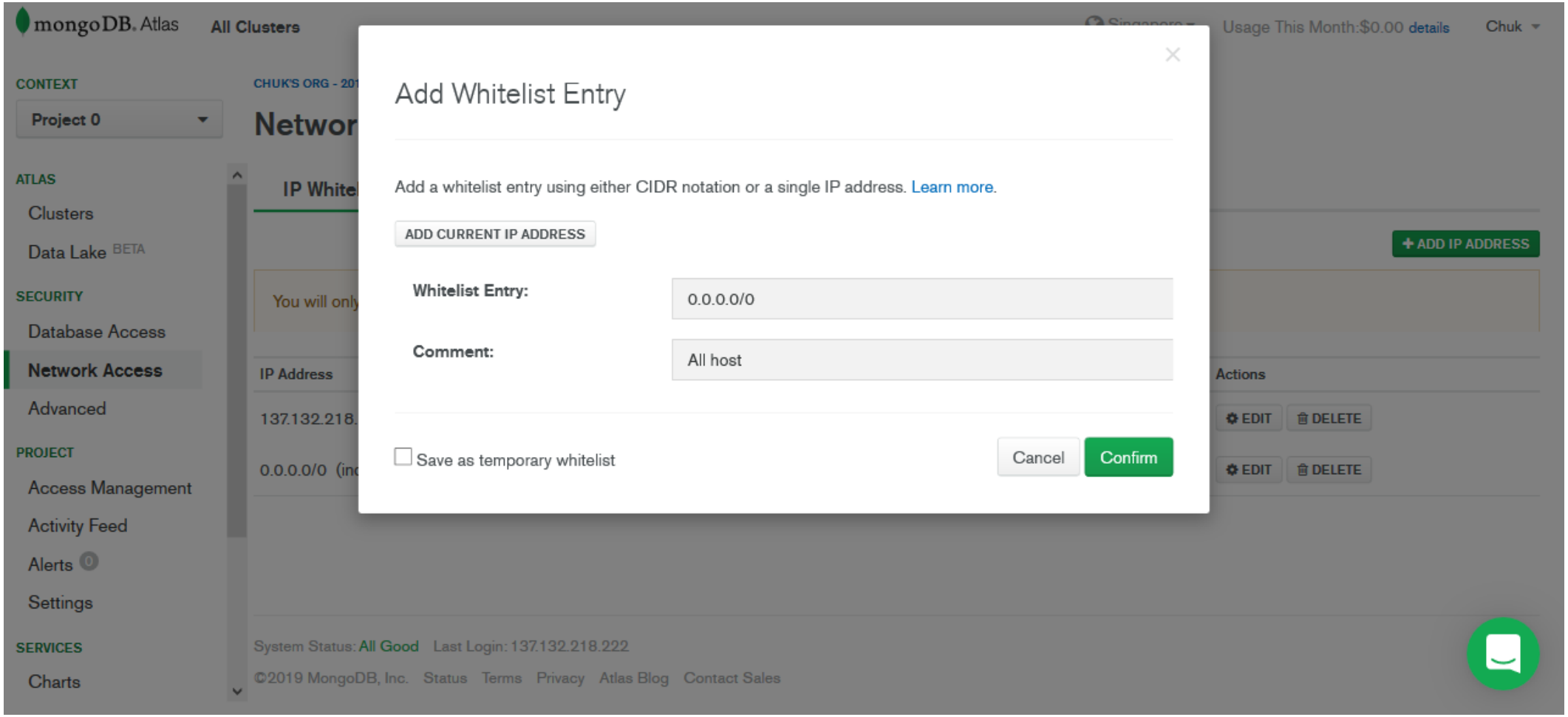
Create Users



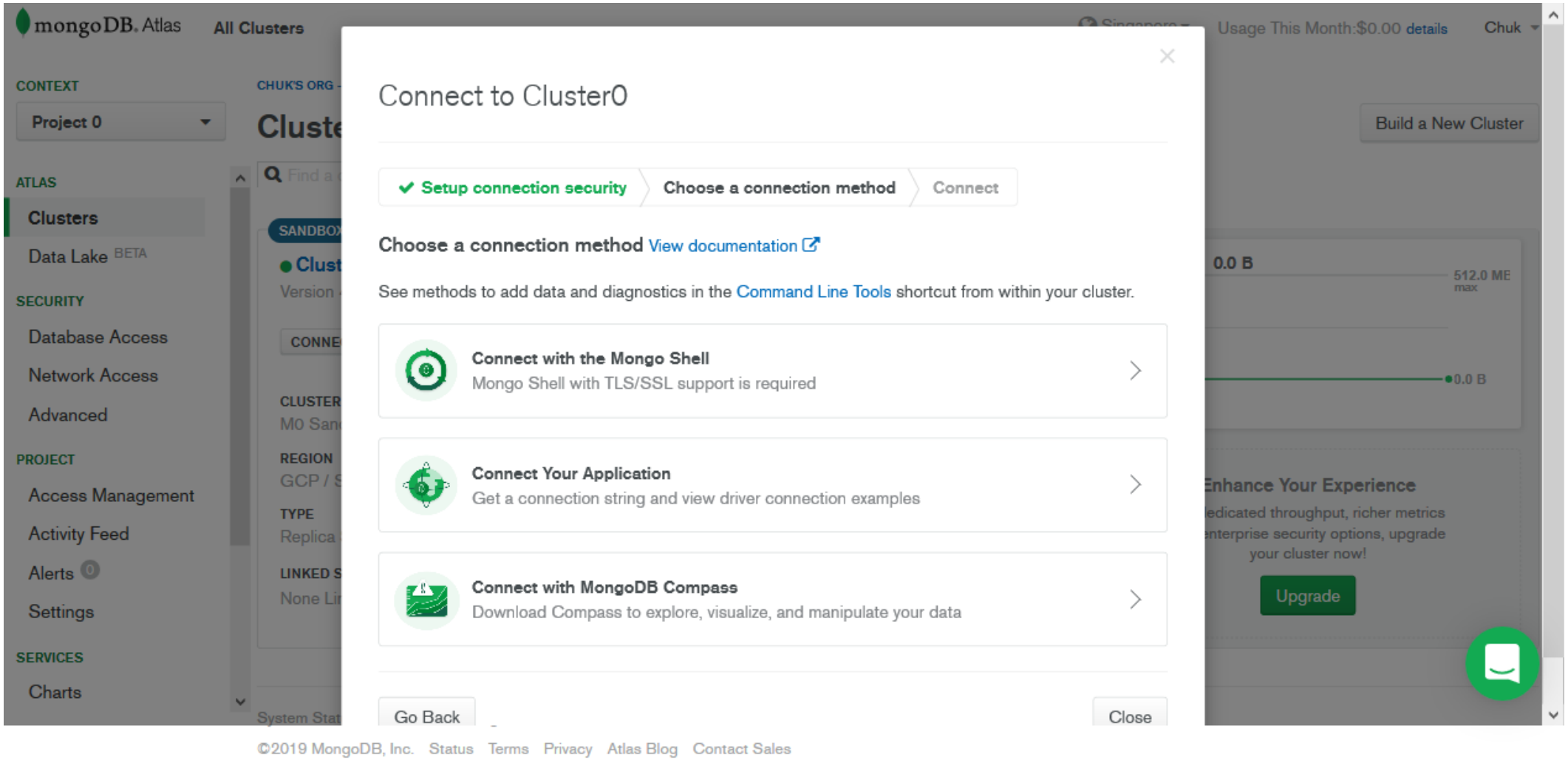
Create User


Add New User
SCRAM Authentication
SCRAM is MongoDB's default authentication method.
bwayne
e.g. new-user_31
SHOW
Autogenerate Secure Password
User Privileges
Atlas admin
Read and write to any database
Only read any database
Select Custom Role
Add Default Privileges
Save as temporary user
Cancel
Add User

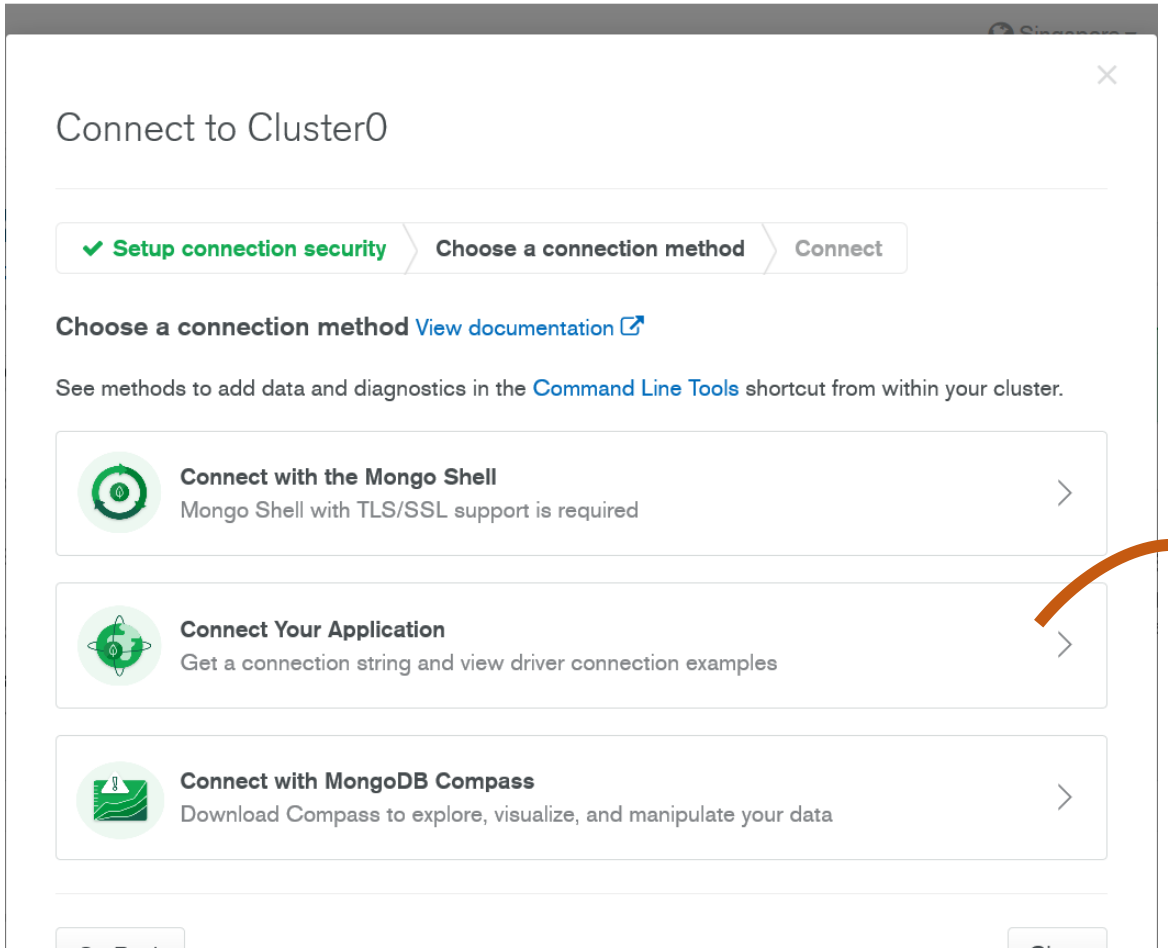
Network Access

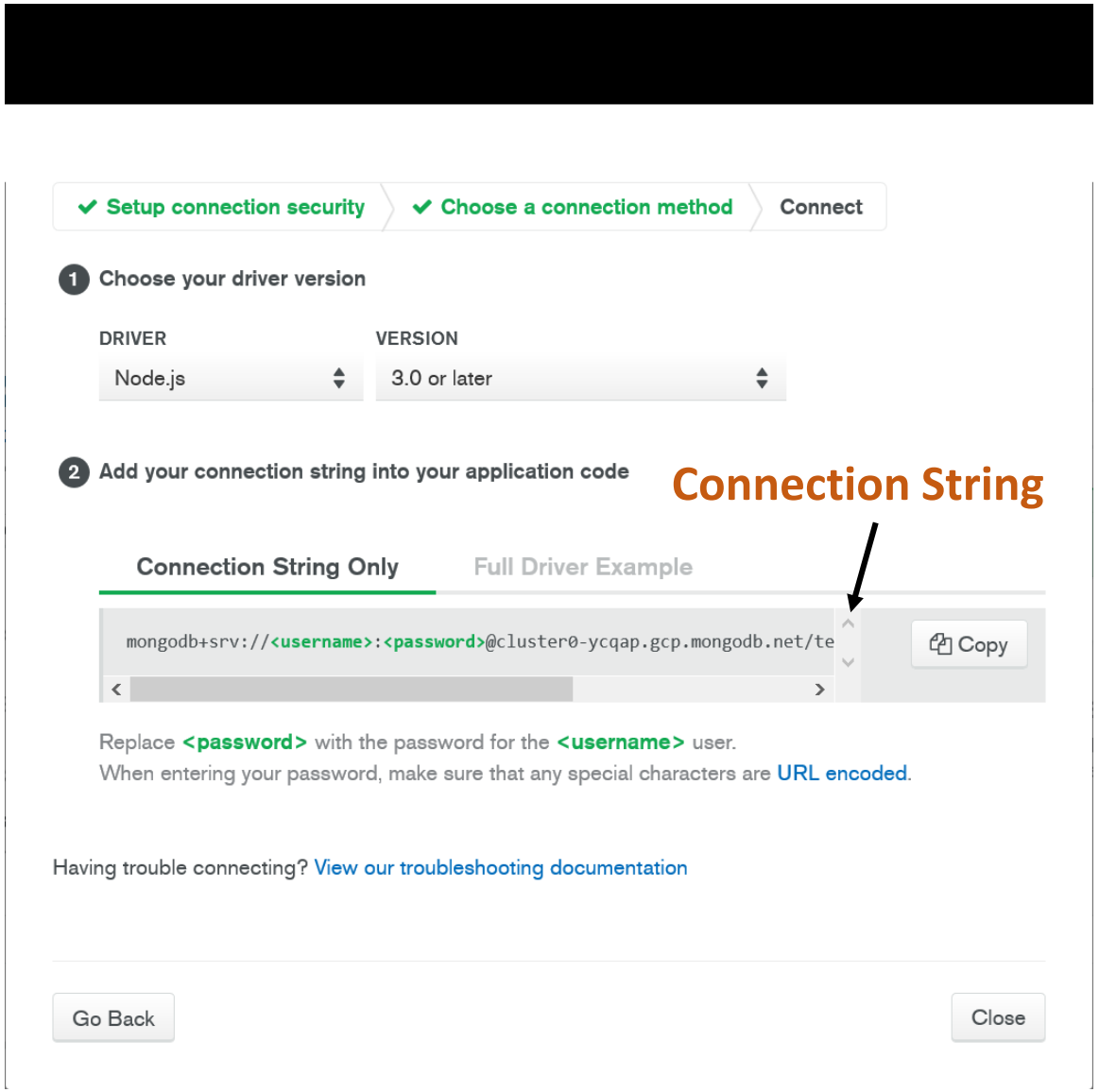


Accessing the Cluster Programmatically

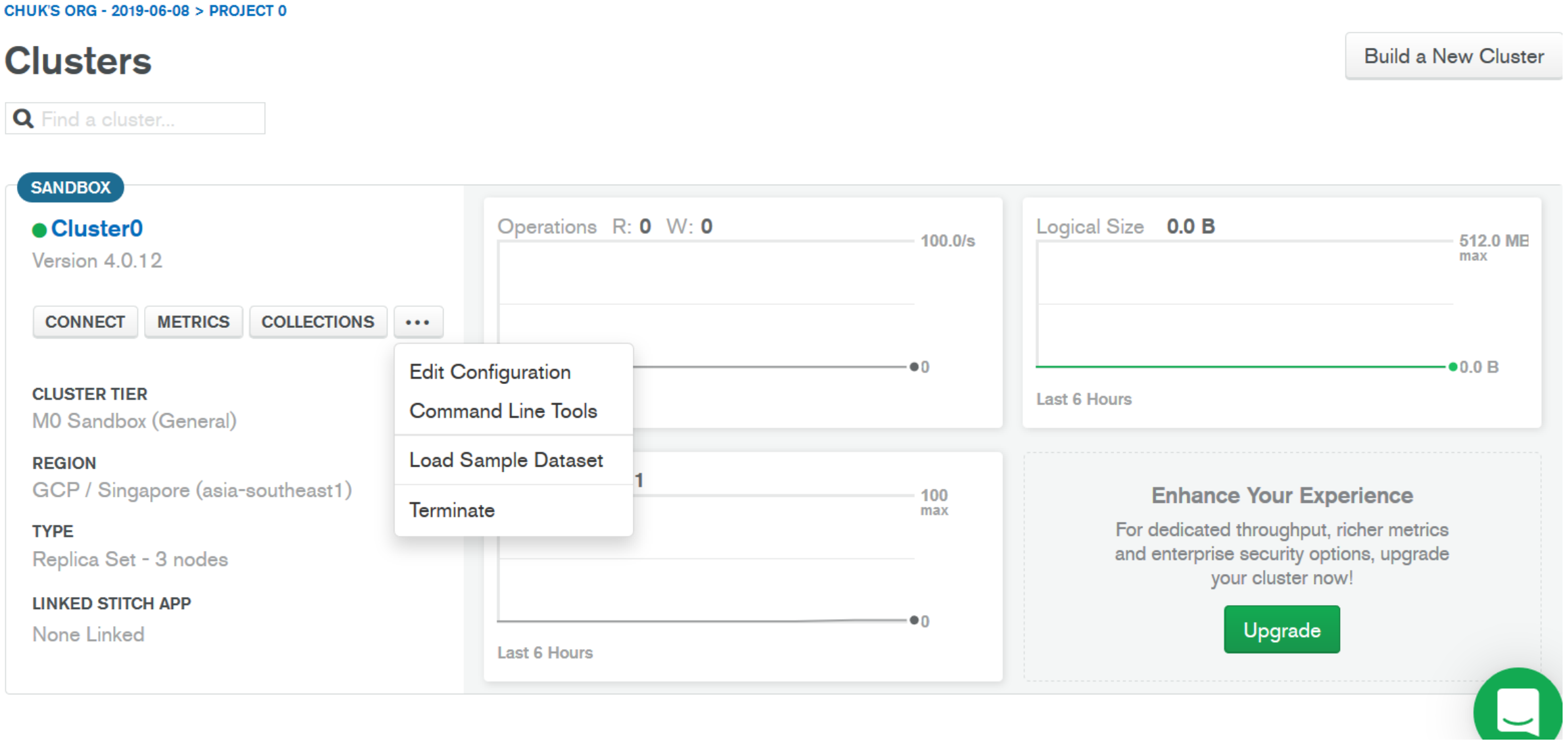


Connection String

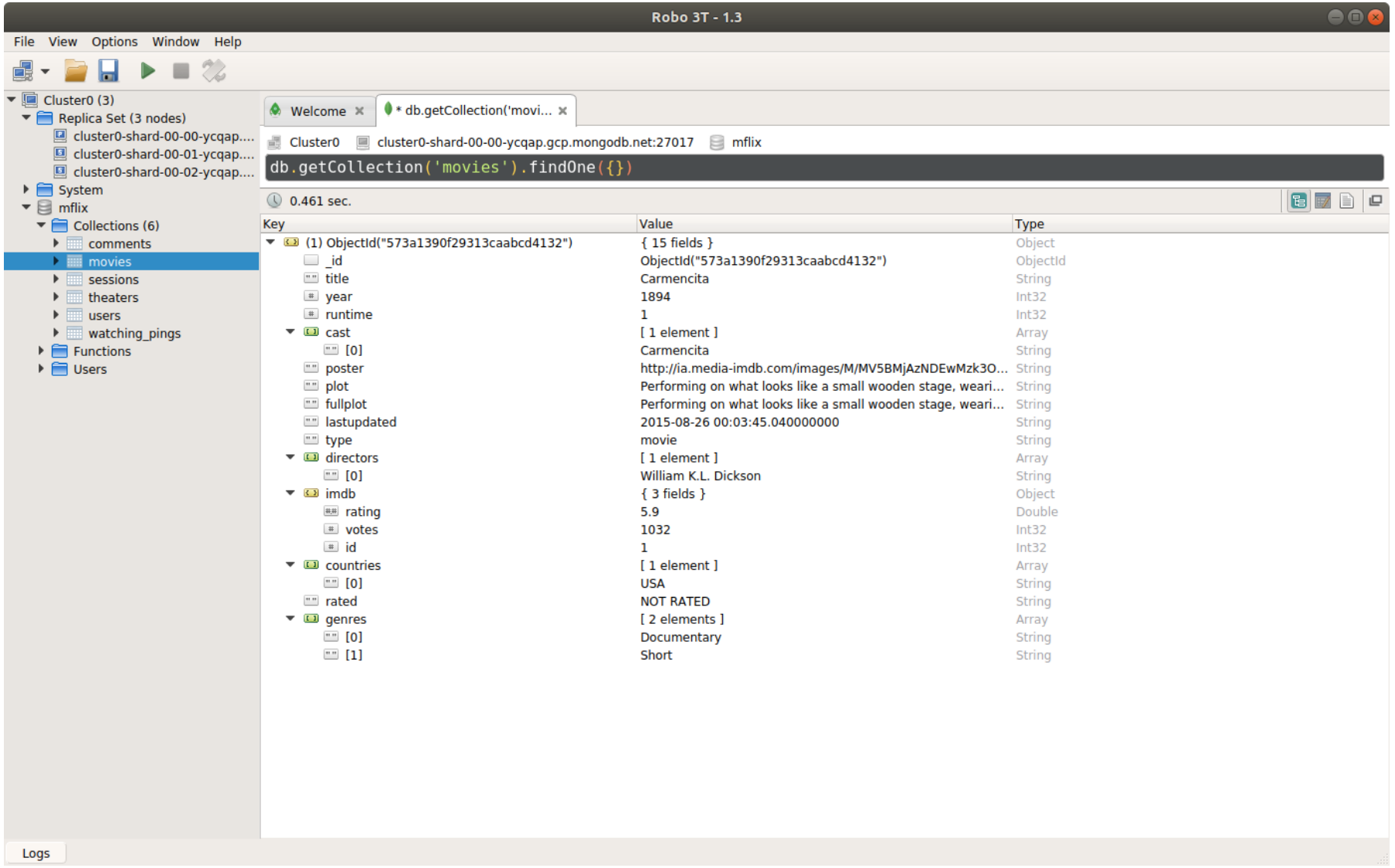




Accessing the Cluster via Command Line Tools



MongoDB GUI Client



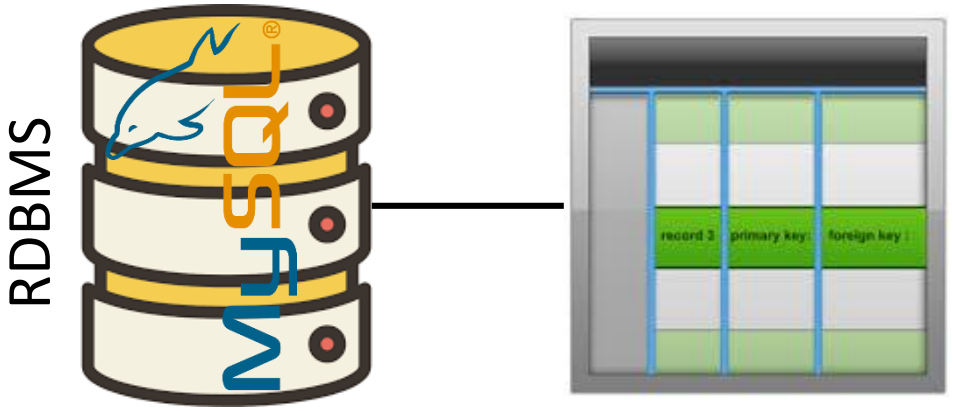
MongoDB - How its Different from RDBMS

- All documents has a primary key called `_id`
  - Either assigned by the user
  - Or assigned by MongoDB when document is added to collection
- Collections are created on demand
  - When the first document is added to a collection
- Documents in a collection do not have to be identical
  - Same schema
- Mongo allows you to query over non existence collections and attributes
  - Will not flag as error

Schemas

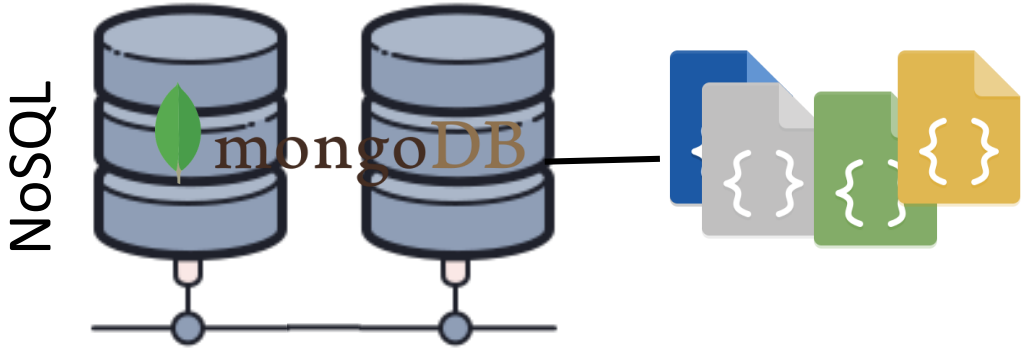
- Explicit schema
- Schema on write
  - Inserts must conform to the predefine schema

```
create table customer (
   cust_id int primary key,
   name varchar(64),
   ...
)
```



- Implicit schema
- Schema on read
  - Find out about the schema when a document is read

```
db.customer.insert({
   _id: ObjectId("abc123"),
   name: "Fred",
   ...
})
```



MongoDB Shell Basics

- Connecting to cluster

```
mongo "mongodb+srv://<cluster_name>.gcp.mongodb.net" --username <user_name>
```

- List databases

```
show databases
```

- Select a database
  - The selected database is held in a variable call `db`

```
use <database_name>
```

- List collections

```
show collections
```

Connecting to MongoDB

Dependencies | **ADD DEPENDENCIES...** CTRL + B

Spring Boot DevTools DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data MongoDB NOSQL

Store data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

Add Spring Data MongoDB dependency

MongoTemplate

- SpringBoot's API to query and mutate Mongo database
  - Like `JDBCTemplate` and `RedisTemplate`, wrapper over Mongo library
  - Will use it for CUD operations, will use R with the native Mongo library

```
@Bean
public MongoTemplate mongoTemplate() {
  return new MongoTemplate(mongoClient(), "movies");
}
```

Database name

Create the template with the client and the database name

An instance of `MongoClient`

Configuring MongoDB Connection

```
@Configuration
public class AppConfig {
   @Value("${MONGO_URL}")
   private String connectionString;

   private MongoClient client = null;

   @Bean
   public MongoClient mongoClient() {
      if (null == client)
         client = MongoClients.create(connectionString);
      return client;
   }

   @Bean
   public MongoTemplate mongoTemplate() {
      return new MongoTemplate(mongoClient(), "movies");
   }
}
```

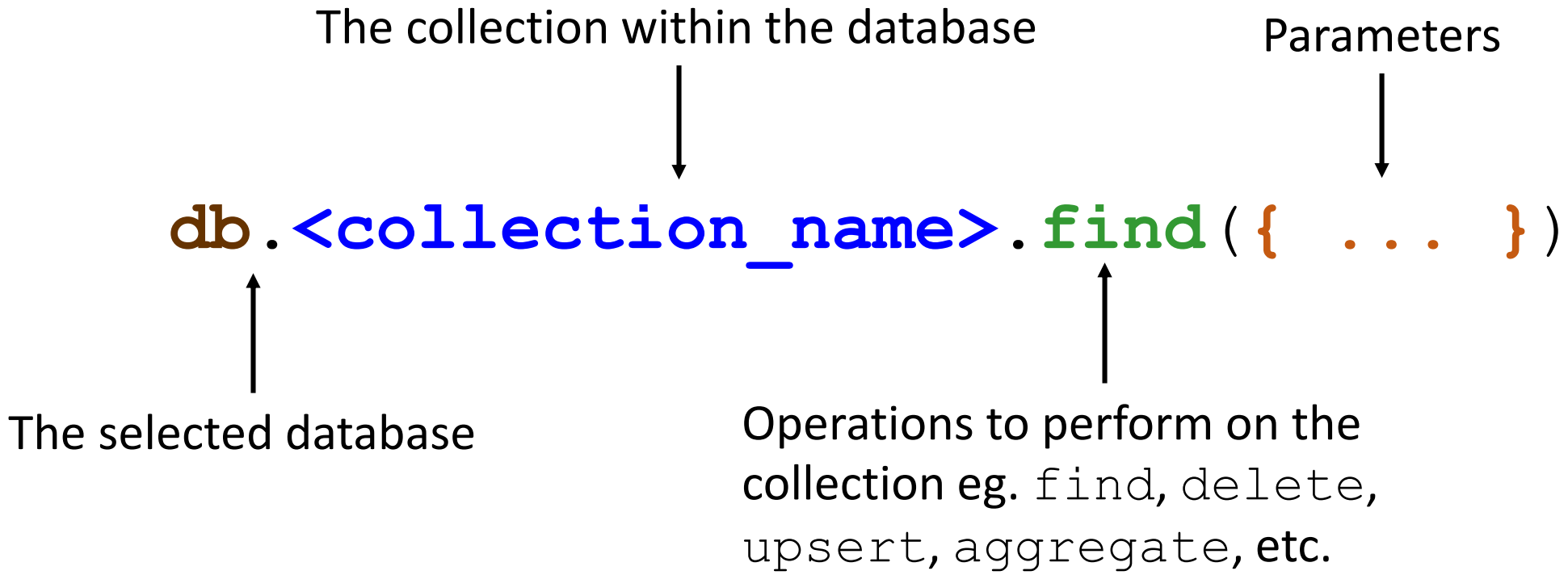
Get an existing database or create a new database

Create an instance of `MongoClient`
Communicates with MongoDB
Singleton

Database name

Create the template with the client and the database name

Database Operations



Sample Data - tv_shows

```
{
   _id: ObjectId(“abc123”),
   title: “Dark Crystal”,
   year: 1982,
   plot: “On another planet in the distant past, a Gelfling embarks on a
quest to find the missing shard of a magical crystal, and so restore
order to his world.”,
   directors: [ “Jim Henson”, “Frank Oz” ],
   imdb: {
      rating: 7.2,
      votes: 56021,
      id: “tt0083791”
   },
   gross: 41373966
   image: “https://....jpg”,
   genres: [ “Adventure”, “Family”, “Fantasy” ],
   rating: “PG”
}
```

Read - `find`

```
db.tv_shows.find({ title: 'Dark Crystal' })
```

Find the document with the value ‘Dark Crystal’ in `title` attribute

```
db.tv_shows.find({
  language: 'English',
  type: 'Scripted'
})
```

Find the English scripted TV shows

Read - find

```
@Autowired
private MongoTemplate mongoTemplate;

Criteria criterial = Criteria.where("title").is("Dark Crystal");
Query query = Query.query(criteria);

List<Document> result = mongoTemplate.find(
   query, Document.class, "tv_shows");


Criteria criterial = Criteria
   .where("language").is("English")
   .and("type").is("Scripted");
Query query = Query.query(criteria);

List<Document> result = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

Create a predicate

Use the predicate to create a filter

Search documents from the tv_shows collection with the filter. Returns a list of `org.bson.Document` objects

Predicate with multiple and conditions

Working with `Document`

- List of `getXXX(String fieldName)` method to return attributes from JSON document

```
getString("title"),getDouble("rating"),
getBoolean("married"),getDate("dob")
```

- To JSON string

```
return ResponseEntity.ok(doc.toJson());
```

- From JSON string

```
JsonObject json = ...
Document doc = Document.parse(json.toString());
```

Read - `find`

```
db.tv_shows.find({
  title: {
    $regex: "crystal",
    $optionals: "i"
  }
})
```

Find the document with the pattern 'crystal' in `title` attribute. The `i` denotes case insensitive search

```
db.tv_shows.find({
  title: { $regex: "crystal", $optionals: "i" },
  year: 1982
})
```

Find the document with the pattern 'crystal' in `title` attribute and with the value 1982 in `year` attribute

Read - find

```
@Autowired
private MongoTemplate mongoTemplate;

Criteria criterial = Criteria.where("title")
   .regex("crystal", "i");
Query query = Query.query(criteria);

List<Document> result = mongoTemplate.find(
   query, Document.class, "tv_shows");


Criteria criterial = Criteria.andOperator(
   Criteria.where("title").regex("crystal", "i"),
   Criteria.where("year").is(1982)
);
Query query = Query.query(criteria);

List<Document> result = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

An alternative for constructing an and predicate

Read - find

Default type for document id

```
{
   _id: ObjectId("abc123"),
   title: "Dark Crystal",
   ...
}
```

```
db.tv_shows.find({ _id: 'abc123' })
```

Will not return any result

```
db.tv_shows.find({ _id: ObjectId('abc123') })
```

Read - find

Create an instance of `ObjectId`

```
public Optional<Document> findTVShowById(String id) {
   ObjectId docId = new ObjectId(id);
   return Optional.ofNullable(
      mongoTemplate.findById(docId, Document.class, "tv_shows");
   );
}
```

Either finds the document or returns null

Addressing Attributes

- Single attribute
  - `title`
- Attribute in an embedded document
  - `imdb.votes`
- Projected field
  - Value of a particular field in the current document
  - Eg. `$gross`
  - Used in aggregation

```
{
   _id: ObjectId("abc123"),
   title: "Dark Crystal",
   year: 1982,
   plot: "On another ..",
   directors: [ "Jim Henson", "Frank Oz" ],
   imdb: {
      rating: 7.2,
      votes: 56021,
      id: "tt0083791"
   },
   gross: 41373966
   image: {
      medium: "https://....jpg",
      original: "https://....jpg"
   }
   genres: [ "Adventure", "Family", "Fantasy" ],
   rating: "PG"
}
```

List of Operators

- Logical
  - `$and`, `$or`, `$not`, `$nor`
- Comparison
  - `$eq`, `$neq`, `$gt`, `$gte`, `$lt`, `$lte`, `$in`, `$nin`
- Element query
  - `$exists`, `$type`
- See https://docs.mongodb.com/manual/reference/operator/

Read - find

```
db.tv_shows.find({ year: { $gte: 1984 } })
```

Find the document where the `year` attribute is greater than or equal to 1984

```
db.tv_shows.find({
   $and: [
      { year: { $gte: 1984 } },
      { "imdb.rating": { $gt: 5.5 } }
   ]
})
```

Find all documents where the IMBD `rating` is greater than 5.5 starting from the `year` 1984

Read - find

```
Query query = Query.query(
   Criteria.where("year").gte(1984)
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

```
Query query = Query.query(
   Criteria.andOperator(
      Criteria.where("year").gte(1984),
      Criteria.where("imdb.rating").gt(5.5)
   )
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

Read - `find`

```
db.tv_shows.find({
   genre: {
      $in: [ "Drama", "Horror", "Adventure" ]
   }
})
```

Find the documents where the `genre` is one of the value in the array

```
db.tv_shows.find({
   awards: { $exists: true }
})
```

Find the document which has the `awards` attribute

Read - find

```
Query query = Query.query(
   Criteria.where("genre")
      .in("Drama", "Horror", "Adventure")
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

```
Query query = Query.query(
   Criteria.where("awards")
      .exists(true)
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

Read - find

```
{ ... genres: [ "Science-Fiction", "Drama", "Crime" ] },
{ ... genres: [ "Drama", "Crime" ] },
{ ... genres: [ "Thriller", "Drama", "Crime" ] },
```

```
db.inventory.find({ "genres": "Drama" })
```

Find any document with `Drama`

```
db.inventory.find({ "genres":
  { $in: [ "Science-Fiction", "Crime" ] } })
```

Find any document with `Science-Fiction` or `Crime`

```
db.inventory.find({ "genres":
  { $all: [ "Science-Fiction", "Crime" ] } })
```

Find any document with `Science-Fiction` and `Crime`

Read - find

```
Query query = Query.query(
   Criteria.where("genre").in("Drama")
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");

Query query = Query.query(
   Criteria.where("genre").in(List.of("Science-Fiction", "Crime"))
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");

Query query = Query.query(
   Criteria.where("genre").all(List.of("Science-Fiction", "Crime"))
);
List<Document> results = mongoTemplate.find(
   query, Document.class, "tv_shows");
```

Read - Miscellaneous

```
db.tv_shows.distinct('rated')
```

Returns an array of unique values for that field

```
db.tv_shows.find({ year: { $gte: 2000 } }).count()
```

Count the number for documents returned by the query

Read - Miscellaneous

```
List<String> ratings = mongoTemplate.findDistinct(
  new Query(), "rated", "tv_shows", String.class);
```

The type of the `rated` property

```
Query query = Query.query(
  Criteria.where("year").gte(2000)
);
long count = mongoTemplate.count(query, "tv_shows");
```

Paginating Results

- Limiting the number of result

```
db.tv_shows.find({
   awards: { $exists: true }
}).limit(10)
```

Display only 10 documents

- Return the result starting from a specified document

```
db.tv_shows.find({
   awards: { $exists: true }
}).skip(10)
```

Only return the results starting from the 11th document

```
db.tv_shows.find({
   awards: { $exists: true }
}).skip(10)
.limit(5)
```

Return documents from 11 to 15
Note order of `skip()` and `limit()` is unimportant. MongoDB will always perform `skip()` first before `limit()`

Paginating Results

```
Query query = Query.query(
   Criteria.where("awards").exists(true)
).limit(10);
```

```
Query query = Query.query(
   Criteria.where("awards").exists(true)
).skip(10);
```

```
Query query = Query.query(
   Criteria.where("awards").exists(true)
).limit(10).skip(10);
```

Sorting Results

- Sorting the result
  - -1 ascending, 1 - descending

```
db.tv_shows.find({
   awards: { $exists: true }
}).sort({ title: -1 })
```

Sorting on some attributes may require an index
`db.tv_shows.createIndex({ title: 1 })`
Indices will be covered in the next section

```
db.tv_shows.find({
   awards: { $exists: true }
}).skip(10)
.limit(5)
.sort({ title: -1, gross: 1 })
```

Sort the result based on the following 2 attributes

MongoDB perform the `skip()`, `limit()` `sort()` in the order show regardless of the order in which they are called

Sorting Results

```
Query query = Query.query(
    Criteria.where("awards").exists(true)
).with(
   Sort.by(Sort.Direction.ASC, "title")
);
```

```
Query query = Query.query(
    Criteria.where("awards").exists(true)
).with(
   Sort
      .by(Sort.Direction.ASC, "title")
      .by(Sort.Direction.DESC, "gross")
).skip(10).limit(5);
```

Projections

- By default queries returns all fields
- Projection allows you to specify to MongoDB to only return a subset of fields

```
db.tv_shows.find(
   { title: /.*kill.*/, year: 1982, },
   { _id: 0, title: 1, plot: 1, gross: 1, image: 1 }
)
```

it's $regex but doesn't let you use $options

Will always return `_id` by default. Need to explicitly disable it if you're not interested in the `_id`

Will only return these attributes

Projections

```
Query query = Query.query(
   Criteria.where("title").regex(".*kill.*")
      .and("year").is(1982)
);
query.fields()
   .exclude("_id")
   .include("title", "plot", "gross", "image");

List<Document> results = mongoTemplate.query(
   query, Document.class, "tv_shows");
```

Predicate Summary

Property to apply the operation to

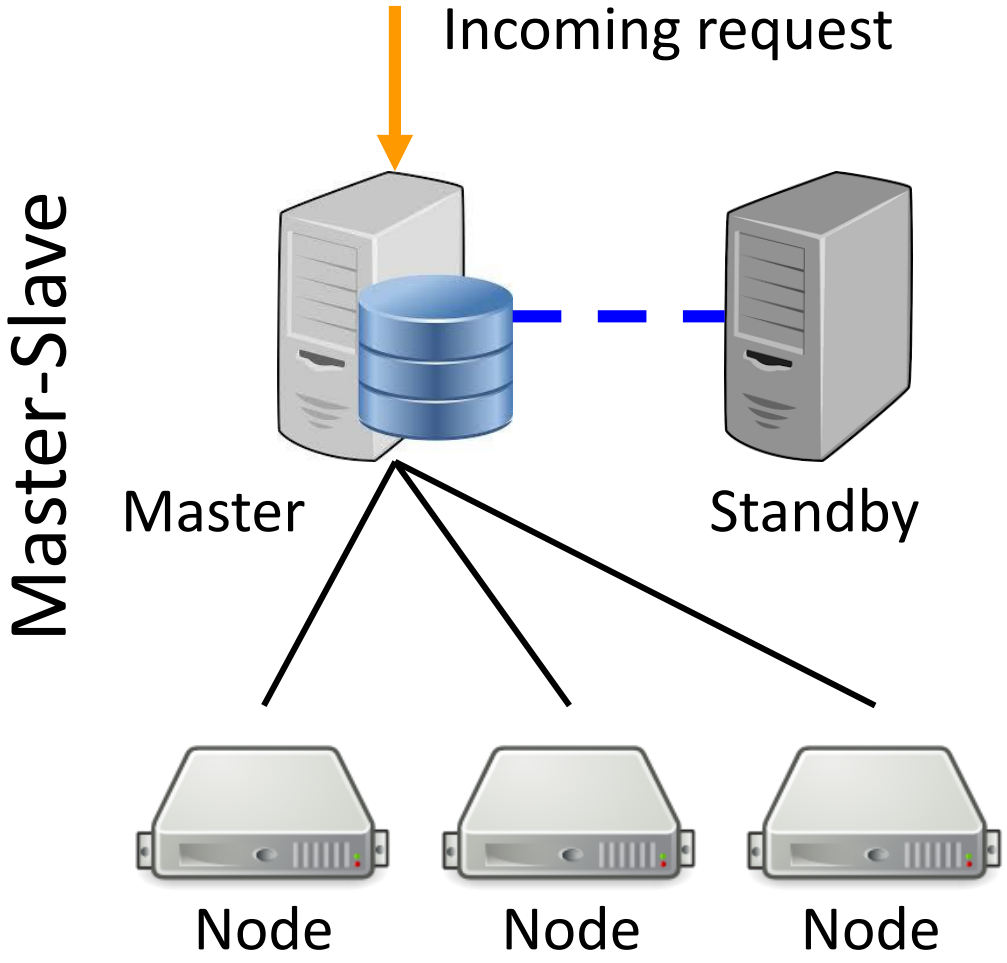
```
Criteria.where("property_name").<operation>()
```

```
Criteria.where("title").is("Titanic")
Criteria.where("year").gt(2000)
Criteria.where("genre")
   .in("Horror", "Family")
Criterial.where("awards").exists(true)
Criteria.andOperator(List.of(
   Criteria.where("title").regex(".*land.*),
   Criteria.where("rated").is("PG-13"))
)
```

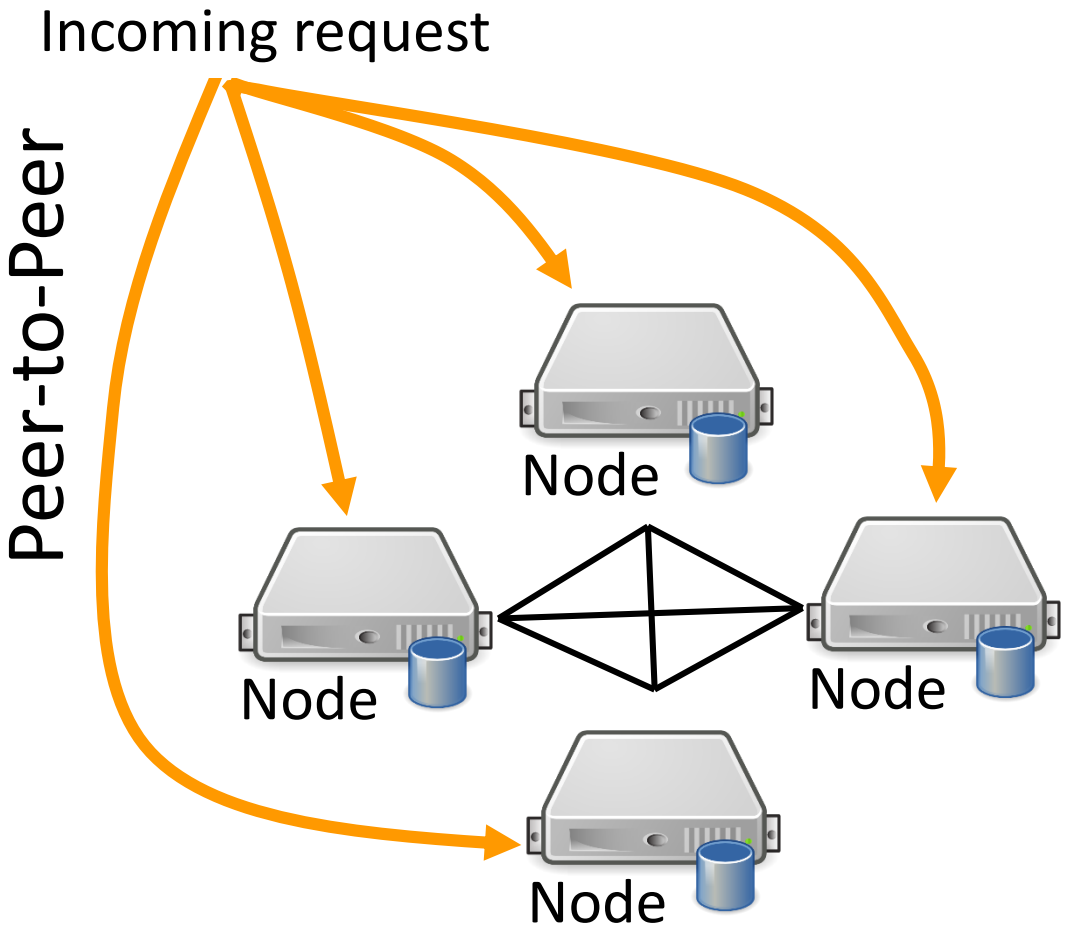
```
{ title: "Titanic" }
{ year: { $gt: 2000 } }
{ genre: { $in: [ "Horror", "Family" ] }
{ awards: { $exists: true } }
{ $and: [
   { title: { $regex: ".*land.*" } },
   { rated: "PG-13" }
]}
```

Unused

Architecture



Relational - ACID



NoSQL - BASE