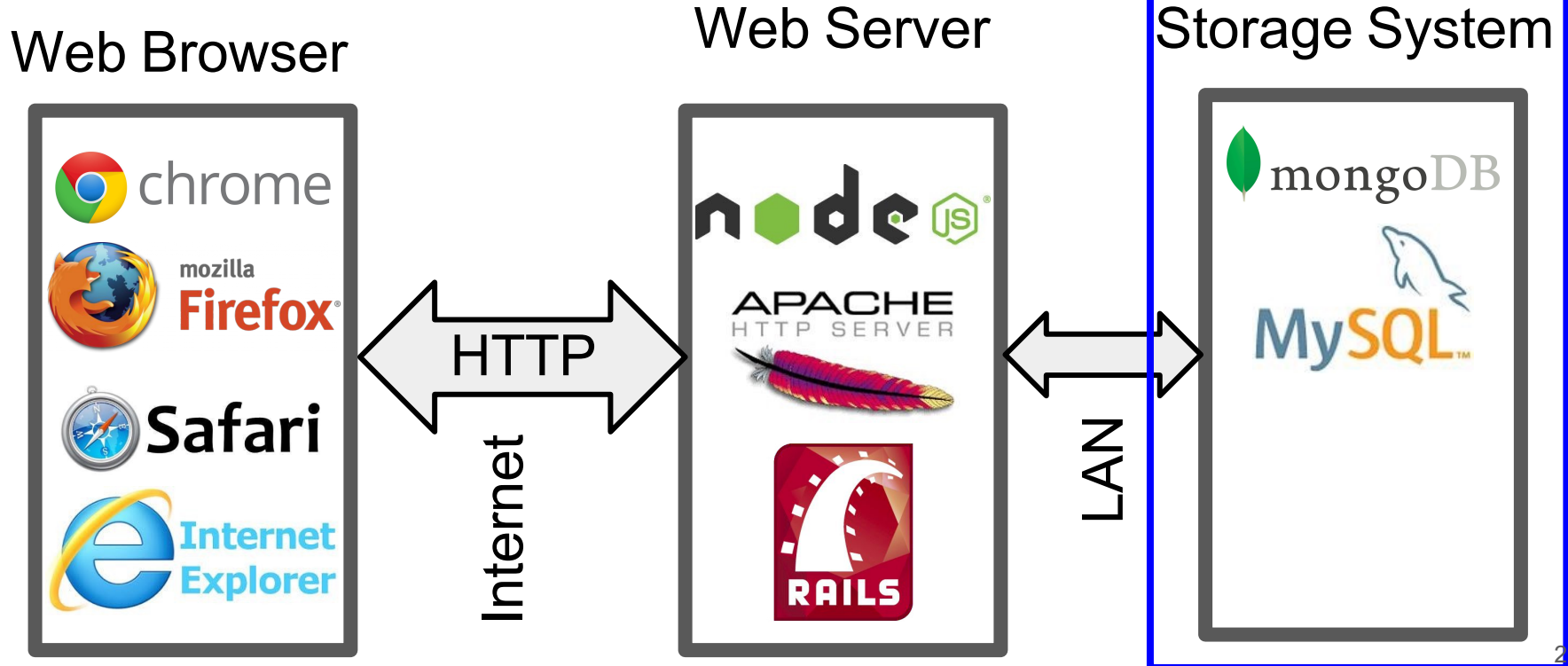


Storage Tier

Web Application Architecture



Web App Storage System Properties

- Always available - Fetch correct app data, store updates
 - Even if many request come in concurrently - Scalable
 - From all over the world
 - Even if pieces fail - Reliable / fault tolerant
- Provide a good organization of storing an application data
 - Quickly generate the model data of a view
 - Handle app evolving over time
- Good software engineering: Easy to use and reason about

Relational Database System

- Early on many different structures file system, objects, networks, etc.
 - The database community decided the answer was the **relational** model
 - Many in the community still think it is.
- Data is organized as a series of **tables** (also called **relations**)

A table is made of up of **rows** (also called **tuples** or **records**)

A row is made of a fixed (per table) set of typed **columns**

- String: VARCHAR(20)
- Integer: INTEGER
- Floating-point: FLOAT, DOUBLE
- Date/time: DATE, TIME, DATETIME
- Others

Database Schema

Schema: The structure of the database

- The table names (e.g. User, Photo, Comments)
- The names and types of table columns
- Various optional additional information (constraints, etc.)

Example: User Table

Column types

ID - INTEGER

first_name - VARCHAR(20)

last_name - VARCHAR(20)

location - VARCHAR(20)

ID	first_name	last_name	location
1	Ian	Malcolm	Austin, TX
2	Ellen	Ripley	Nostromo
3	Peregrin	Took	Gondor
4	Rey	Kenobi	D'Qar
5	April	Ludgate	Awnee, IN
6	John	Ousterhout	Stanford, CA

Structured Query Language (SQL)

- Standard for accessing relational data
 - Sweet theory behind it: **relational algebra**
- Queries: the strength of relational databases
 - Lots of ways to extract information
 - You specify what you want
 - The database system figures out how to get it efficiently
 - Refer to data by contents, not just name

SQL Example Commands

```
CREATE TABLE Users (  
    id INT AUTO_INCREMENT,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20),  
    location VARCHAR(20));
```

```
INSERT INTO Users (  
    first_name,  
    last_name,  
    location)  
VALUES  
('Ian',  
'Malcolm',  
'Austin, TX');
```

```
DELETE FROM Users WHERE  
    last_name='Malcolm';
```

```
UPDATE Users  
    SET location = 'New York, NY'  
    WHERE id = 2;
```

```
SELECT * FROM Users;
```

```
SELECT * from Users WHERE id = 2;
```


Keys and Indexes

Consider a model fetch: `SELECT * FROM Users WHERE id = 2`

Database could implement this by:

1. **Scan** the Users table and return all rows with id=2
2. Have built an **index** that maps id numbers to table rows. Lookup result from index.

Uses **keys** to tell database that building an index would be a good idea

Primary key: Organize data around accesses

`PRIMARY KEY(id)` on a `CREATE` table command

Secondary key: Other indexes (`UNIQUE`)

Object Relational Mapping (ORM)

- Relational model and SQL was a bad match for Web Applications
 - Object versus tables
 - Need to evolve quickly
- 2nd generation web frameworks (Rails) handled mapping objects to SQL DB
- Rail's Active Record
 - Objects map to database records
 - One class for each table in the database (called **Models** in Rails)
 - Objects of the class correspond to rows in the table
 - Attributes of an object correspond to columns from the row
- Handled all the schema creation and SQL commands behind object interface

NoSQL - MongoDB

- Using SQL databases provided reliable storage for early web applications
- Led to new databases that matched web application object model
 - Known collectively as NoSQL databases
- MongoDB - Most prominent NoSQL database
 - Data model: Stores **collections** containing **documents** (JSON objects)
 - Has expressive query language
 - Can use **indexes** for fast lookups
 - Tries to handle scalability, reliability, etc.

Schema enforcement

- JSON blobs provide super flexibility but not what is always wanted
 - Consider: `<h1>Hello {person.informalName}</h1>`
 - Good: `typeof person.informalName == 'string'` and `length < something`
 - Bad: Type is 1GB object, or undefined, or null, or ...
- Would like to enforce a **schema** on the data
 - Can be implemented as **validators** on mutating operations
- Mongoose - Object Definition Language (ODL)
 - Take familiar usage from ORMs and map it onto MongoDB
 - Exports **Persistent Object** abstraction
 - Effectively masks the lower level interface to MongoDB with something that is friendlier

Using: `let mongoose = require('mongoose');`

1. Connect to the MongoDB instance

```
mongoose.connect('mongodb://localhost/dbname');
```

2. Wait for connection to complete: Mongoose exports an EventEmitter

```
mongoose.connection.on('open', function () {  
    // Can start processing model fetch requests  
});
```

```
mongoose.connection.on('error', function (err) { });
```

Can also listen for connecting, connected, disconnecting, disconnected, etc.

Mongoose: Schema define collections

Schema assign property names and their types to collections

String, Number, Date, Buffer, Boolean

Array - e.g. comments: [ObjectId]

ObjectId - Reference to another object

Mixed - Anything

```
var userSchema = new mongoose.Schema({  
  first_name: String,  
  last_name: String,  
  emailAddresses: [String],  
  location: String  
});
```

Schema allows secondary indexes and defaults

- Simple index

```
first_name: {type: 'String', index: true}
```

- Index with unique enforcement

```
user_name: {type: 'String', index: {unique: true} }
```

- Defaults

```
date: {type: Date, default: Date.now }
```

Secondary indexes

- Performance and space trade-off
 - Faster queries: Eliminate scans - database just returns the matches from the index
 - Slower mutating operations: Add, delete, update must update indexes
 - Uses more space: Need to store indexes and indexes can get bigger than the data itself
- When to use
 - Common queries spending a lot of time scanning
 - Need to enforce uniqueness

Mongoose: Make Model from Schema

- A **Model** in Mongoose is a constructor of objects - a collection May or may not correspond to a model of the MVC

```
let User = mongoose.model('User', userSchema);
```

Exports a **persistent** object abstraction

- Create objects from Model

```
User.create({ first_name: 'Ian', last_name: 'Malcolm'}, doneCallback);  
function doneCallback(err, newUser) {  
  assert (!err);  
  console.log('Created object with ID', newUser._id);  
}
```

Model used for querying collection

- Returning the entire User collection

```
User.find(function (err, users) { /*users is an array of objects*/ });
```

- Returning a single user object for user_id

```
User.findOne({_id: user_id}, function (err, user) { /* ... */ });
```

- Updating a user object for user_id

```
User.findOne({_id: user_id}, function (err, user) {  
    // Update user object - (Note: Object is "special")  
    user.save();  
});
```

Other Mongoose query operations - query builder

```
let query = User.find({});
```

- Projections

```
query.select("first_name last_name").exec(doneCallback);
```

- Sorting

```
query.sort("first_name").exec(doneCallback);
```

- Limits

```
query.limit(50).exec(doneCallback);
```

```
query.sort("-location").select("first_name").exec(doneCallback);
```

Deleting objects from collection

- Deleting a single user with id `user_id`

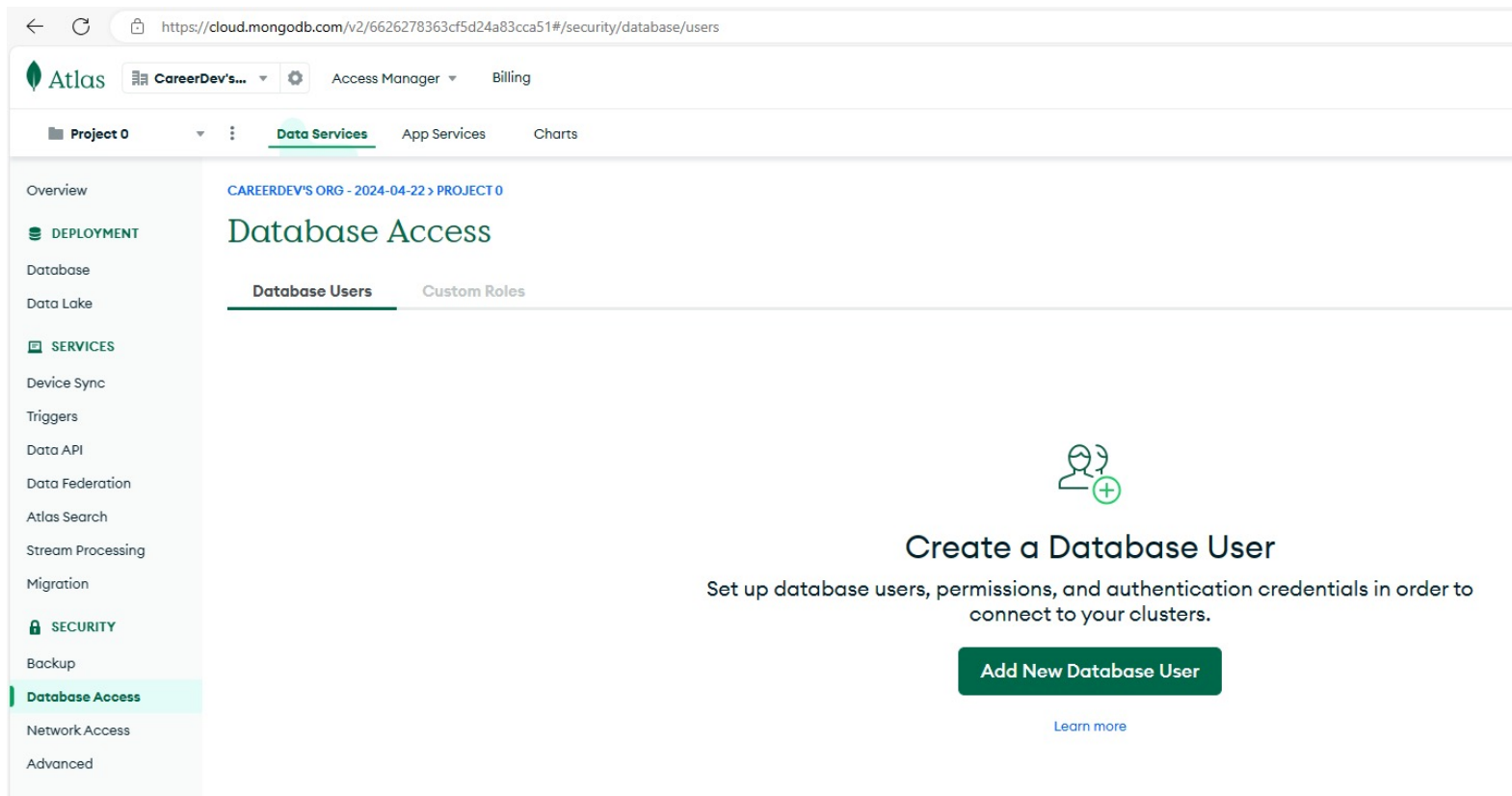
```
User.remove({_id: user_id}, function (err) { } );
```

- Deleting all the User objects

```
User.remove({}, function (err) { } );
```

Simple Blog with MongoDB

Login Mongo Atlas and create a database user



The screenshot shows the MongoDB Atlas web interface. The browser address bar displays the URL: `https://cloud.mongodb.com/v2/6626278363cf5d24a83cca51#/security/database/users`. The top navigation bar includes the Atlas logo, a dropdown menu for 'CareerDev's...', and links for 'Access Manager' and 'Billing'. Below this, a secondary navigation bar shows 'Project 0' and tabs for 'Data Services' (active), 'App Services', and 'Charts'. The left sidebar contains a navigation menu with sections: 'Overview', 'DEPLOYMENT' (with sub-items: Database, Data Lake), 'SERVICES' (with sub-items: Device Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration), 'SECURITY' (with sub-items: Backup, Database Access - highlighted), 'Network Access', and 'Advanced'. The main content area is titled 'Database Access' and has a sub-header 'CAREERDEV'S ORG - 2024-04-22 > PROJECT 0'. Below this, there are two tabs: 'Database Users' (active) and 'Custom Roles'. The central part of the page features a large green icon of a person with a plus sign, followed by the heading 'Create a Database User'. Below the heading is a descriptive text: 'Set up database users, permissions, and authentication credentials in order to connect to your clusters.' At the bottom of this section is a prominent green button labeled 'Add New Database User'. A small blue link 'Learn more' is positioned at the very bottom right of the page.

← ↻ 🔒 <https://cloud.mongodb.com/v2/6626278363cf5d24a83cca51#/security/database/users>

Atlas CareerDev's... Access Manager Billing

Project 0 Data Services App Services Charts

Overview

DEPLOYMENT

- Database
- Data Lake

SERVICES

- Device Sync
- Triggers
- Data API
- Data Federation
- Atlas Search
- Stream Processing
- Migration


SECURITY

- Backup
- Database Access**
- Network Access
- Advanced

CAREERDEV'S ORG - 2024-04-22 > PROJECT 0

Database Access

Database Users Custom Roles



Create a Database User

Set up database users, permissions, and authentication credentials in order to connect to your clusters.

[Add New Database User](#)

[Learn more](#)

Login Mongo Atlas and create a database user

Create a database user to grant an application or user access to databases and collections in your clusters in this Atlas project. Granular access control can be configured with default privileges or custom roles. You can grant access to an Atlas project or organization using the corresponding [Access Manager](#).

Authentication Method

Password

Certificate

AWS IAM
(MongoDB 4.4 and up)

PREVIEW
Federated Auth
(MongoDB 7.0 and up)

MongoDB uses [SCRAM](#) as its default authentication method.

Password Authentication

testuser

..... SHOW

🔍 Autogenerate Secure Password 📋 Copy

Database User Privileges

Configure role based access control by assigning database user a mix of one built-in role, multiple custom roles, and multiple specific privileges. A user will gain access to all actions within the roles assigned to them, not just the actions those roles share in common. **You must choose at least one role or privilege.** [Learn more about roles.](#)

Built-in Role

0 SELECTED ^

Select one [built-in role](#) for this user.

Select Role

Atlas admin

Read and write to any database

Only read any database

or select a custom role in the [Custom Roles](#) tab.

Build a database

Project 0

Data Services

App Services

Charts

Overview

DEPLOYMENT

Database

Data Lake

SERVICES

Device Sync

Triggers

Data API

Data Federation

Atlas Search

Stream Processing

Migration

SECURITY

Backup


Database Access

Network Access

Advanced

CAREERDEV'S ORG - 2024-04-22 > PROJECT 0

Database Deployments



Create a database

Choose your cloud provider, region, and specs.

Build a Database

Once your database is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#).

Create a cluster

Deploy your database

Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.

☐ **M10** **\$0.10/hour**
For production applications with sophisticated workload requirements.

STORAGE	RAM	vCPU
10 GB	2 GB	2 vCPUs

☐ **Serverless**
For application development and testing, or workloads with variable traffic.

STORAGE	RAM	vCPU
Up to 1TB	Auto-scale	Auto-scale

☒ **M0** **Free**
For learning and exploring MongoDB in a cloud environment.

STORAGE	RAM	vCPU
512 MB	Shared	Shared

✓ **Free forever!** Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Name

You cannot change the name once the cluster is created.

☒ Automate security setup ⓘ

☒ Add sample dataset ⓘ

Provider



Region

Hong Kong (ap-east-1) ★

★ Recommended ⓘ

Low carbon emissions ⓘ

I'll do this later

Go to Advanced Configuration

Create Deployment

Connect a user to the cluster

The screenshot displays the Atlas web interface. At the top, the 'Atlas' logo is on the left, and 'CareerDev's...' with a dropdown and a settings gear icon are on the right. Below this, 'Access Manager' and 'Billing' links are visible. The main navigation bar includes 'Project 0', 'Data Services' (highlighted), 'App Services', and 'Charts'. A left sidebar lists navigation options: 'Overview' (selected), 'DEPLOYMENT' (with sub-items 'Database' and 'Data Lake'), 'SERVICES' (with sub-items 'Device Sync', 'Triggers', 'Data API', 'Data Federation', 'Atlas Search', 'Stream Processing', and 'Migration'), and 'SECURITY' (with a sub-item 'Quickstart'). The main content area is titled 'Overview' and shows 'CAREERDEV'S ORG - 2024-04-22 > PROJECT 0'. Under the 'Database Deployments' section, a 'Cluster0' tab is active. A green checkmark message states 'Sample Dataset successfully loaded!'. Below this, three buttons are present: 'Connect' (circled in red), 'View info', and 'Edit configuration'. At the bottom of the cluster card, there is a '+ Add Tag' button. To the right of the cluster card, there are two action buttons: 'Browse collections' and 'View monitoring', each with a right-pointing arrow.

Connect a user to the cluster

Connect to Cluster0



Connect to your application



Drivers

Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)



Access your data through tools



Data Explorer

Browse your Atlas collections without leaving the UI



Compass

Explore, modify, and visualize your data with MongoDB's GUI



Shell

Quickly add & update data using MongoDB's Javascript command-line interface



Connect a user to the cluster

Connect to Cluster0



Connecting with MongoDB Driver

1. Select your driver and version

We recommend installing and using the latest driver version.

Driver	Version
Node.js ▼	5.5 or later ▼

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

[View MongoDB Node.js Driver installation instructions.](#)

3. Add your connection string into your application code

☐ View full code sample

```
mongodb+srv://testuser:<password>@cluster0.m30zoib.mongodb.net/?  
retryWrites=true&w=majority&appName=Cluster0
```

Replace **<password>** with the password for the **testuser** user. Ensure any option params are [URL encoded](#).

Create databases and collections

Overview

DEPLOYMENT

Database

Data Lake

SERVICES

Device Sync

Triggers

Data API

Data Federation

Atlas Search

Stream Processing

Migration

SECURITY

Quickstart

Backup

Database Access

Network Access

CAREERDEV'S ORG - 2024-04-22 > PROJECT 0

Database Deployments

Find a database deployment...

Sample dataset successfully loaded. Access it in Collections or by connecting with the MongoDB Shell.

Cluster0

Connect

View Monitoring

Browse Collections

...

Visualize Your Data

Build dashboards and charts, and embed them in your apps with MongoDB Charts.

Dismiss

Explore Charts

R 0

W 0

Last 11 minutes

39.5/s

Connections 1.0

Last 16 minutes

8.0

In 117.8 B/s

Out 910.0 B/s

Last 11 minutes

223.1 KB/s

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL	ATLAS SEARCH
7.0.8	AWS / Hong Kong (ap-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Connect	Create Index

Create databases and collections

The screenshot displays the MongoDB Atlas web interface. On the left, a sidebar contains navigation links: Overview, DEPLOYMENT, Database (highlighted), Data Lake, SERVICES, Device Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration, SECURITY, Quickstart, Backup, Database Access, and Network Access. The main header shows the breadcrumb 'CAREERDEV'S ORG - 2024-04-22 > PROJECT 0 > DATABASES' and the 'Cluster0' logo. Below the header, a series of tabs includes Overview, Real Time, Metrics, Collections (selected), Atlas Search, Performance Advisor, Online Archive, and Cmd Line 1. The 'Collections' tab shows 'DATABASES: 1' and 'COLLECTIONS: 6'. A red circle highlights the '+ Create Database' button. Below it is a search bar labeled 'Search Namespaces'. The left sidebar under the 'Database' section shows a tree view with 'sample_mflix' expanded, listing collections: comments, embedded_movies, movies, sessions, theaters, and users. The right pane displays details for the 'sample_mflix.comments' collection, including storage and index sizes, document counts, and tabs for Find, Indexes, Schema Anti-Patterns, Aggregation, and Search Indexes. A query filter box is present, and the 'QUERY RESULTS' section shows a single document with fields like _id, name, email, movie_id, text, and date.

Overview

DEPLOYMENT

Database

Data Lake

SERVICES

Device Sync

Triggers

Data API

Data Federation

Atlas Search

Stream Processing

Migration

SECURITY

Quickstart

Backup

Database Access

Network Access

CAREERDEV'S ORG - 2024-04-22 > PROJECT 0 > DATABASES

Cluster0

Overview Real Time Metrics Collections Atlas Search Performance Advisor Online Archive Cmd Line 1

DATABASES: 1 COLLECTIONS: 6

+ Create Database

Search Namespaces

sample_mflix

comments

embedded_movies

movies

sessions

theaters

users

sample_mflix.comments

STORAGE SIZE: 5.7MB LOGICAL DATA SIZE: 11.14MB TOTAL DOCUMENTS: 41079 INDEXES TOTAL SIZE: 1.12MB

Find Indexes Schema Anti-Patterns Aggregation Search Indexes

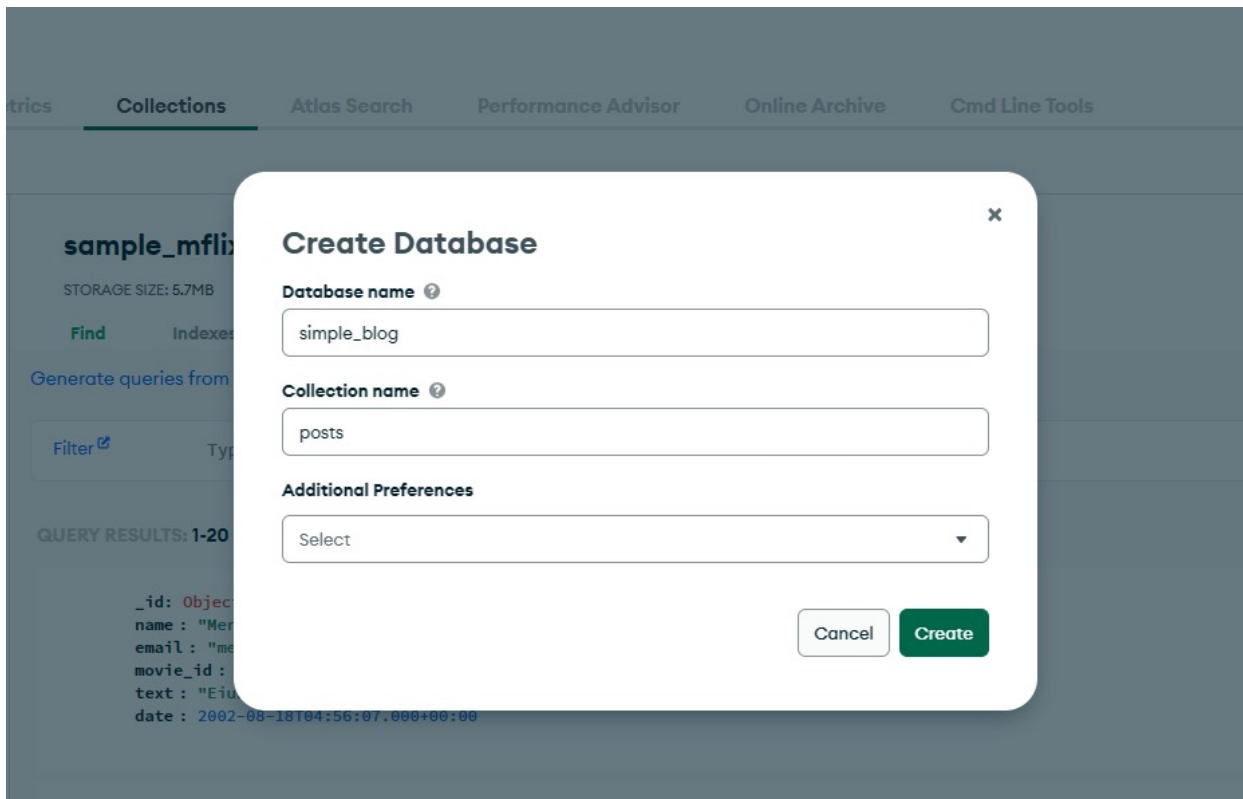
Generate queries from natural language in Compass

Filter Type a query: { field: 'value' }

QUERY RESULTS: 1-20 OF MANY

```
{
  "_id": ObjectId('5a9427648b0beebe69579e7'),
  "name": "Mercedes Tyler",
  "email": "mercedes_tyler@fakegmail.com",
  "movie_id": ObjectId('573a1390f29313caabcd4323'),
  "text": "Eius veritatis vero facilis quaerat fuga temporibus. Praesentium exped...",
  "date": 2002-08-18T04:56:07.000+00:00
}
```

Create databases and collections



The screenshot shows the MongoDB Atlas interface with a modal dialog for creating a new database. The background interface includes a top navigation bar with tabs: Metrics, Collections (selected), Atlas Search, Performance Advisor, Online Archive, and Cmd Line Tools. On the left, there's a sidebar with 'sample_mflix' database info (Storage Size: 5.7MB), a 'Find' button, 'Indexes' link, 'Generate queries from' link, a 'Filter' button with a help icon, and a 'Type' dropdown. The main area shows 'QUERY RESULTS: 1-20' and a snippet of a document: `{ "_id": ObjectId("..."), "name": "Megan", "email": "megan@...", "movie_id": "Eid...", "text": "Eid...", "date": "2002-08-18T04:56:07.000+00:00" }`.

Create Database [Close]

Database name [?]

Collection name [?]

Additional Preferences

[Dropdown Arrow]

Create databases and collections

The screenshot displays the MongoDB Atlas web interface. On the left is a navigation sidebar with sections: Overview, DEPLOYMENT, Database (highlighted), Data Lake, SERVICES, Device Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration, SECURITY, Quickstart, and Backup. The main content area is titled 'CAREERDEV'S ORG - 2024-04-22 > PROJECT 0 > DATABASES' and features the ClusterO logo. Below the logo are tabs for Overview, Real Time, Metrics, Collections (selected), Atlas Search, Performance Advisor, and Online Archive. A summary bar indicates 'DATABASES: 2' and 'COLLECTIONS: 7'. On the left of the main panel, there is a '+ Create Database' button and a 'Search Namespaces' input field. Below these, a tree view shows 'sample_mflix' and 'simple_blog' (expanded) with 'posts' listed underneath. The right side of the panel is dedicated to the 'simple_blog.posts' collection, showing its storage size (4KB), logical data size (0B), total documents (0), and total index size (4KB). It includes links for Find, Indexes, Schema Anti-Patterns (with a notification icon), Aggregation, and Search Indexes. A link to 'Generate queries from natural language in Compass' is also present. A 'Filter' button and a query input field containing '{ field: 'value' }' are shown. At the bottom, it states 'QUERY RESULTS: 0'.

Overview

DEPLOYMENT

Database

Data Lake

SERVICES

Device Sync

Triggers

Data API

Data Federation

Atlas Search

Stream Processing

Migration

SECURITY

Quickstart

Backup

CAREERDEV'S ORG - 2024-04-22 > PROJECT 0 > DATABASES

ClusterO

Overview Real Time Metrics **Collections** Atlas Search Performance Advisor Online Archive

DATABASES: 2 COLLECTIONS: 7

+ Create Database

Search Namespaces

sample_mflix

simple_blog

posts

simple_blog.posts

STORAGE SIZE: 4KB LOGICAL DATA SIZE: 0B TOTAL DOCUMENTS: 0 INDEXES TOTAL SIZE: 4KB

Find Indexes Schema Anti-Patterns 1 Aggregation Search Indexes

Generate queries from natural language in Compass

Filter Type a query: { field: 'value' }

QUERY RESULTS: 0

Integrating Mongoose with Express

- Install Mongoose: `npm install mongoose`
- Create a file in the root folder and name it `.env`. Create a variable `DB_URL` and assign the connection string to it:

```
DB_URL=mongodb+srv://<username>:<passwd>@cluster0.suftrmo.mongodb.net/simple_blog?retryW  
rites=true&w=majority&appName=Cluster0
```

- Create a `db` directory, and add a `dbConnect.js` file with the following code:

```
const mongoose = require("mongoose");  
require("dotenv").config();  
  
async function dbConnect() {  
  mongoose .connect(process.env.DB_URL)  
    .then(() => {  
      console.log("Successfully connected to MongoDB Atlas!");  
    })  
    .catch((error) => {console.error(error);})  
};  
module.exports = dbConnect;
```

Integrating Mongoose with Express

- In the `index.js` file, import and call the `connectDB` function to create a connection to the MongoDB instance

```
const express = require("express");
...
const dbConnect = require("../db/dbConnect"); // <- add this

const app = express();
...

dbConnect(); // <- add this

app.use(cors());
app.use(express.json()); // <- add this
app.use("/api", PostRouter); // <- add this
app.listen(8080, () => {
  console.log("server listening on port 8080");
});
```

Creating Schema and Models with Mongoose

- In **db** directory, create a **postModel.js** file with the following code:

```
const mongoose = require("mongoose");
const PostSchema = new mongoose.Schema({
  slug: {
    type: String, required: [true, "Please provide slug"], unique: [true, "Slug Exist"],
  },
  title: {
    type: String, required: [true, "Please provide a title!"],
  },
  description: {
    type: String, required: [true, "Please provide a description!"],
  },
});

module.exports = mongoose.model.Posts || mongoose.model("Posts", PostSchema);
```

Adding Data to MongoDB with the POST Method

- In the root directory, create a **routes** directory, and add an **PostRouter.js** file with the following code:

```
const express = require("express");
const Post = require("../db/postModel");
const router = express.Router();

router.post("/post", async (request, response) => {
  const post = new Post(request.body);
  try {
    await post.save();
    response.send(post);
  } catch (error) {
    response.status(500).send(error);
  }
});

module.exports = router;
```

Return All Posts Using the GET Method

- In the `PostRouter.js` file, create the GET endpoint:

```
const express = require("express");
...
router.post("/post", async (request, response) => {
  ...
});

router.get("/posts", async (request, response) => {
  try {
    const posts = await Post.find({});
    response.send(posts);
  } catch (error) {
    response.status(500).send({ error });
  }
});

module.exports = router;
```

GET Endpoint Returning a Single Article By ID

- In the `PostRouter.js` file, create the endpoint as follows:

```
const express = require("express");
...
router.post("/post", async (request, response) => {
  ...
});
...
router.get("/post/:slug", async (request, response) => {
  try {
    const post = await Post.findOne({ slug: request.params.slug });
    response.send(post);
  } catch (error) {
    response.status(500).send({ error });
  }
});

module.exports = router;
```

Updating Existing MongoDB Data Using PATCH

- In the `PostRouter.js` file, add the following code to create the `PATCH` endpoint:

```
const express = require("express");
...
router.patch("/post/:slug", async (request, response) => {
  try {
    const post = await Post.findByIdAndUpdate(request.params.slug,
      request.body,);
    await post.save();
    response.send(post);
  } catch (error) {
    response.status(500).send({ error });
  }
});

module.exports = router;
```

Deleting A Post Using DELETE Method

- In the `PostRouter.js` file, add the following code to create the **DELETE** endpoint:

```
const express = require("express");
...
router.delete("/post/:slug", async (request, response) => {
  try {
    const post = await Post.findByIdAndDelete(request.params.slug);
    if (!post) {
      return response.status(404).send("Post wasn't found");
    }
    response.status(204).send();
  } catch (error) {
    response.status(500).send({ error });
  }
});

module.exports = router;
```