How to Use MongoDB Transactions in Node.js

Lauren Schaefer

December 11, 2019 | Updated: February 19, 2020
#Node.js #JavaScript / node.js #transactions

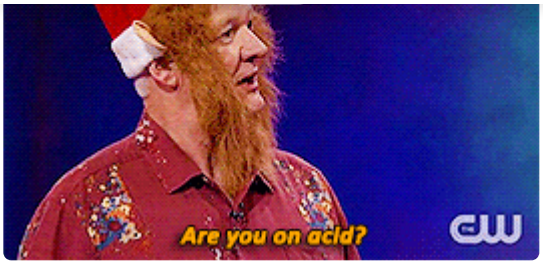
Developers who move from relational databases to MongoDB commonly ask, “Does MongoDB support ACID transactions? If so, how do you create a transaction?” The answer to the first question is, “Yes!”

Beginning in 4.0 MongoDB added support for multi-document ACID transactions, and beginning in 4.2 MongoDB added support for distributed ACID transactions. If you’re not familiar with what ACID transactions are or if you should be using them in MongoDB, check out my earlier post on the subject.

For those of you just joining us in this Quick Start with MongoDB and Node.js series, welcome! We began by walking through how to connect to MongoDB and perform each of the CRUD—create, read, update, and delete—operations. Then we jumped into more advanced topics like the aggregation framework.

The code we write today will use the same structure as the code we built in the first post in the series, so, if you have any questions about how to get started or how the code is structured, head back to that first post.

Now let’s dive into that second question developers ask--let’s discover how to create a transaction!



> Get started with an M0 cluster on Atlas today. It's free forever, and it's the easiest way to try out the steps in this blog series.

Creating an Airbnb Reservation

As you may have experienced while working with MongoDB, most use cases do not require you to use multi-document transactions. When you model your data using our rule of thumb **Data that is accessed together should be stored together**, you'll find that you rarely need to use a multi-document transaction. In fact, I struggled a bit to think of a use case for the Airbnb dataset that would require a multi-document transaction.

After a bit of brainstorming, I came up with a somewhat plausible example. Let's say we want to allow users to create reservations in the `sample_airbnb` database.

We could begin by creating a collection named `users`. We want users to be able to easily view their reservations when they are looking at their profiles, so we will store the reservations as embedded documents in the `users` collection. For example, let's say a user named Leslie creates two reservations. Her document in the `users` collection would look like the following:

```
{
  "_id": {"$oid":"5dd589544f549efc1b0320a5"},
  "email": "leslie@example.com",
  "name": "Leslie Yepp",
  "reservations":  [
    {
      "name":"Infinite Views",
      "dates": [
         {"$date": {"$numberLong":"1577750400000"}},
         {"$date": {"$numberLong":"1577836800000"}}
      ],
```

```
    },
    {
      "name": "Lovely Loft",
       "dates": [
         {"$date": {"$numberLong": "1585958400000"}}
      ],
      "pricePerNight": {"$numberInt":"210"}
      "breakfastIncluded":false
    }
  ]
}
```

When browsing Airbnb listings, users need to know if the listing is already booked for their travel dates. As a result we want to store the dates the listing is reserved in the `listingsAndReviews` collection. For example, the "Infinite Views" listing that Leslie reserved should be updated to list her reservation dates.

```
{
  "_id": {"$oid":"5dbc20f942073d6d4dabd730"},
  "name":"Infinite Views",
  "summary":"Modern home with infinite views from the infinity pool",
  "property_type": "House",
  "bedrooms": {"$numberInt": "6"},
  "bathrooms":{"$numberDouble":"4.5"},
  "beds":{"$numberInt":"8"},
  "datesReserved": [
    {"$date": {"$numberLong": "1577750400000"}},
    {"$date": {"$numberLong": "1577836800000"}}
  ]
}
```

Keeping these two records in sync is imperative. If we were to create a reservation in a document in the `users` collection without updating the associated document in the `listingsAndReviews` collection, our data would be inconsistent. We can use a multi-document transaction to ensure both updates succeed or fail together.

Set Up

We'll be using the "Infinite Views" Airbnb listing we created in a previous post in this series. Hop back to the post on Creating Documents if your database doesn't currently have the Infinite Views listing.

The Airbnb sample dataset only has the listingsAndReviews collection by default. To help you quickly create the necessary collection and data, I wrote usersCollection.js. Download a copy of the file, update the `uri` constant to reflect your Atlas connection info, and run it by executing `node usersCollection.js`. The script will create three new users in the `users` collection: Leslie Yepp, April Ludfence, and Tom Haverdodge. If the `users` collection does not already exist, MongoDB will automatically create it for you when you insert the new users. The script also creates an index on the `email` field in the `users` collection. The index requires that every document in the `users` collection has a unique `email`.

Create a Transaction in Node.js

Now that we are set up, let's implement the functionality to store Airbnb reservations.

Get a Copy of the Node.js Template

To make following along with this blog post easier, I've created a starter template for a Node.js script that accesses an Atlas cluster.

1. Download a copy of template.js.
2. Open template.js in your favorite code editor.
3. Update the Connection URI to point to your Atlas cluster. If you're not sure how to do that, refer back to the first post in this series.
4. Save the file as `transaction.js`.

You can run this file by executing `node transaction.js` in your shell. At this point, the file simply opens and closes a connection to your Atlas cluster, so no output is expected. If you see DeprecationWarnings, you can ignore them for the purposes of this post.

Create a Helper Function

Let's create a helper function. This function will generate a reservation document that we will use later.

1. Paste the following function in `transaction.js`:

```
        let reservation = {
            name: nameOfListing,
            dates: reservationDates,
        }

        // Add additional properties from reservationDetails to the reservation
        for (let detail in reservationDetails) {
            reservation[detail] = reservationDetails[detail];
        }

        return reservation;
    }
```

To give you an idea of what this function is doing, let me show you an example. We could call this function from inside of `main()`:

```
createReservationDocument("Infinite Views",
    [new Date("2019-12-31"), new Date("2020-01-01")],
    { pricePerNight: 180, specialRequests: "Late checkout", breakfastIncluded: true });
```

The function would return the following:

```
    { name: 'Infinite Views',
        dates: [ 2019-12-31T00:00:00.000Z, 2020-01-01T00:00:00.000Z ],
        pricePerNight: 180,
        specialRequests: 'Late checkout',
        breakfastIncluded: true }
```

Create a Function for the Transaction

Let's create a function whose job is to create the reservation in the database.

1. Continuing to work in transaction.js, create an asynchronous function named `createReservation`. The function should accept a MongoClient, the user's email address, the name of the Airbnb listing, the reservation dates, and any other reservation details as parameters.

2. Now we need to access the collections we will update in this function. Add the following code to `createReservation()`.

```
const usersCollection = client.db("sample_airbnb").collection("users");
const listingsAndReviewsCollection = client.db("sample_airbnb").collection("listingsAndReviews"
```

3. Let's create our reservation document by calling the helper function we created in the previous section. Paste the following code in `createReservation()`.

```
const reservation = createReservationDocument(nameOfListing, reservationDates, reservationDetai
```

4. Every transaction and its operations must be associated with a session. Beneath the existing code in `createReservation()`, start a session.

```
const session = client.startSession();
```

5. We can choose to define options for the transaction. We won't get into the details of those here. You can learn more about these options in the driver documentation. Paste the following beneath the existing code in `createReservation()`.

```
const transactionOptions = {
       readPreference: 'primary',
       readConcern: { level: 'local' },
       writeConcern: { w: 'majority' }
   };
```

6. Now we're ready to start working with our transaction. Beneath the existing code in `createReservation()`, open a `try{}` block, follow it with a `catch{}` block, and finish it with a

```
try {

} catch(e){

} finally {

}
```

7. We can use ClientSession's withTransaction() to start a transaction, execute a callback function, and commit (or abort on error) the transaction. `withTransaction()` requires us to pass a function that will be run inside the transaction. Add a call to `withTransaction()` inside of `try {}`. Let's begin by passing an anonymous asynchronous function to `withTransaction()`.

```
const transactionResults = await session.withTransaction(async () => {}, transactionOptions);
```

8. The anonymous callback function we are passing to `withTransaction()` doesn't currently do anything. Let's start to incrementally build the database operations we want to call from inside of that function. We can begin by adding a reservation to the reservations array inside of the appropriate user document. Paste the following inside of the anonymous function that is being passed to `withTransaction()`.

```
    const usersUpdateResults = await usersCollection.updateOne(
        { email: userEmail },
        { $addToSet: { reservations: reservation } },
        { session });
    console.log(`${usersUpdateResults.matchedCount} document(s) found in the users collection wi
    console.log(`${usersUpdateResults.modifiedCount} document(s) was/were updated to include the
```

9. Since we want to make sure that an Airbnb listing is not double-booked for any given date, we should check if the reservation date is already listed in the listing's `datesReserved` array. If so, we should abort the transaction. Aborting the transaction will rollback the update to the user document we made in the previous step. Paste the following beneath the existing code in the anonymous function.

```
        { session });
    if (isListingReservedResults) {
        await session.abortTransaction();
        console.error("This listing is already reserved for at least one of the given dates. The
        console.error("Any operations that already occurred as part of this transaction will be r
        return;
    }
```

10. The final thing we want to do inside of our transaction is add the reservation dates to the `datesReserved` array in the `listingsAndReviews` collection. Paste the following beneath the existing code in the anonymous function.

```
const listingsAndReviewsUpdateResults = await listingsAndReviewsCollection.updateOne(
        { name: nameOfListing },
        { $addToSet: { datesReserved: { $each: reservationDates } } },
        { session });
 console.log(`${listingsAndReviewsUpdateResults.matchedCount} document(s) found in the listings
 console.log(`${listingsAndReviewsUpdateResults.modifiedCount} document(s) was/were updated to
```

11. We'll want to know if the transaction succeeds. If `transactionResults` is defined, we know the transaction succeeded. If `transactionResults` is undefined, we know that we aborted it intentionally in our code. Beneath the definition of the `transactionResults` constant, paste the following code.

```
if (transactionResults) {
    console.log("The reservation was successfully created.");
} else {
    console.log("The transaction was intentionally aborted.");
}
```

12. Let's log any errors that are thrown. Paste the following inside of `catch(e){ }`:

```
console.log("The transaction was aborted due to an unexpected error: " + e);
```

```
    await session.endSession();
```

At this point, your function should look like the following:

```
async function createReservation(client, userEmail, nameOfListing, reservationDates, reservationDeta

    const usersCollection = client.db("sample_airbnb").collection("users");
    const listingsAndReviewsCollection = client.db("sample_airbnb").collection("listingsAndReviews")

    const reservation = createReservationDocument(nameOfListing, reservationDates, reservationDetail

    const session = client.startSession();

    const transactionOptions = {
        readPreference: 'primary',
        readConcern: { level: 'local' },
        writeConcern: { w: 'majority' }
    };

    try {
        const transactionResults = await session.withTransaction(async () => {

            const usersUpdateResults = await usersCollection.updateOne(
                { email: userEmail },
                { $addToSet: { reservations: reservation } },
                { session });
            console.log(`${usersUpdateResults.matchedCount} document(s) found in the users collectic
            console.log(`${usersUpdateResults.modifiedCount} document(s) was/were updated to include


            const isListingReservedResults = await listingsAndReviewsCollection.findOne(
                { name: nameOfListing, datesReserved: { $in: reservationDates } },
                { session });
            if (isListingReservedResults) {
                await session.abortTransaction();
                console.error("This listing is already reserved for at least one of the given dates.
                console.error("Any operations that already occurred as part of this transaction will
                return;
```

```
                { name: nameOfListing },
                { $addToSet: { datesReserved: { $each: reservationDates } } },
                { session });
            console.log(`${listingsAndReviewsUpdateResults.matchedCount} document(s) found in the li
            console.log(`${listingsAndReviewsUpdateResults.modifiedCount} document(s) was/were updat

        }, transactionOptions);

        if (transactionResults) {
            console.log("The reservation was successfully created.");
        } else {
            console.log("The transaction was intentionally aborted.");
        }
    } catch(e){
        console.log("The transaction was aborted due to an unexpected error: " + e);
    } finally {
        await session.endSession();
    }

}
```

Call the Function

Now that we've written a function that creates a reservation using a transaction, let's try it out! Let's create a reservation for Leslie at the Infinite Views listing for the nights of December 31, 2019 and January 1, 2020.

1. Inside of `main()` beneath the comment that says `Make the appropriate DB calls`, call your `createReservation()` function:

```
    await createReservation(client,
      "leslie@example.com",
      "Infinite Views",
      [new Date("2019-12-31"), new Date("2020-01-01")],
      { pricePerNight: 180, specialRequests: "Late checkout", breakfastIncluded: true });
```

2. Save your file.

```
1 document(s) found in the users collection with the email address leslie@example.cc
1 document(s) was/were updated to include the reservation.
1 document(s) found in the listingsAndReviews collection with the name Infinite View
1 document(s) was/were updated to include the reservation dates.
The reservation was successfully created.
```

Leslie's document in the `users` collection now contains the reservation.

```
{
  "_id": {"$oid":"5dd68bd03712fe11bebfab0c"},
  "email":"leslie@example.com",
  "name":"Leslie Yepp",
  "reservations": [
    {
      "name":"Infinite Views", "dates": [
        {"$date": {"$numberLong":"1577750400000"}},
        {"$date": {"$numberLong":"1577836800000"}}
        ],
      "pricePerNight": {"$numberInt":"180"},
      "specialRequests": "Late checkout",
      "breakfastIncluded":true
    }
  ]
}
```

The "Infinite Views" listing in the `listingsAndReviews` collection now contains the reservation dates.

```
{
  "_id": {"$oid": "5dbc20f942073d6d4dabd730"},
  "name": "Infinite Views",
  "summary": "Modern home with infinite views from the infinity pool",
  "property_type": "House",
  "bedrooms": {"$numberInt":"6"},
  "bathrooms": {"$numberDouble":"4.5"},
  "beds": {"$numberInt":"8"},
  "datesReserved": [
```

```
}
```

Wrapping Up

Today we implemented a multi-document transaction. Transactions are really handy when you need to make changes to more than one document as an all-or-nothing operation.

When you use relational databases, related data is commonly split between different tables in an effort to normalize the data. As a result, transaction usage is fairly common.

When you use MongoDB, data that is accessed together should be stored together. When you model your data this way, you will likely find that you rarely need to use transactions.

This post included many code snippets that built on code written in the first post of this MongoDB and Node.js Quick Start series. To get a full copy of the code used in today's post, visit the Node.js Quick Start GitHub Repo.

Be on the lookout for the next post in this series where we'll discuss change streams.

Additional Resources

- MongoDB official documentation: Transactions
- Blog post: What's the deal with data integrity in relational databases vs MongoDB?
- Informational page with videos and links to additional resources: ACID Transactions in MongoDB
- Whitepaper: MongoDB Multi-Document ACID Transactions

Series Versions

The examples in this article were created with the following application versions:

Component	Version used
MongoDB	4.0
MongoDB Node.js Driver	3.3.2

All posts in the *Quick Start: Node.js and MongoDB* series:

- How to connect to a MongoDB database using Node.js
- How to create MongoDB documents using Node.js
- How to read MongoDB documents using Node.js
- How to update MongoDB documents using Node.js
- How to delete MongoDB documents using Node.js
- Video: How to perform the CRUD operations using MongoDB & Node.js
- How to analyze your data using MongoDB's Aggregation Framework and Node.js
- How to implement transactions using Node.js (this post)
- How to react to database changes with change streams and triggers

← Previous

Next →

How to Build a Customer Success Team in Six Steps: My MongoDB Journey

Our VP of Customer Success, Ozge Tuncel, discusses how to build a successful customer success team.

May 19, 2020

Resources

NoSQL Database Explained

MongoDB Architecture Guide

MongoDB Enterprise Advanced

MongoDB Atlas

MongoDB Stitch

MongoDB Engineering Blog

Education & Support

View Course Catalog

Certification

MongoDB Manual

Installation

FAQ

Popular Topics

Migrate to MongoDB Atlas

Building a REST API with MongoDB Stitch

Ingesting and Visualizing API Data with Stitch and Charts

About

MongoDB, Inc.

Leadership

Press Room

Careers

Contact Us

Legal Notices

Code of Conduct

Follow Us

Facebook

Github

Youtube

Twitter

LinkedIn

Slack

StackOverflow

Get MongoDB Email Updates

Email Address

→