Transactions in Mongoose

Transactions are new in MongoDB 4.0 and Mongoose 5.2.0. Transactions let you execute multiple operations in isolation and potentially undo all the operations if one of them fails. This guide will get you started using transactions with Mongoose.

Getting Started with Transactions

If you haven't already, import mongoose:

```
import mongoose from 'mongoose';
```

To create a transaction, you first need to create a session using or `Mongoose#startSession` or `Connection#startSession()`.

```
// Using Mongoose's default connection
const session = await mongoose.startSession();

// Using custom connection
const db = await mongoose.createConnection(mongodbUri);
const session = await db.startSession();
```

In practice, you should use either the `session.withTransaction()` helper or Mongoose's `Connection#transaction()` function to run a transaction. The `session.withTransaction()` helper handles:

- Creating a transaction
- Committing the transaction if it succeeds
- Aborting the transaction if your operation throws
- Retrying in the event of a transient transaction error.

```
const session = await Customer.startSession();

// The `withTransaction()` function's first parameter is a function
// that returns a promise.
await session.withTransaction(() => {
  return Customer.create([{ name: 'Test' }], { session: session })
});

const count = await Customer.countDocuments();
assert.strictEqual(count, 1);

session.endSession();
```

For more information on the `ClientSession#withTransaction()` function, please see the MongoDB Node.js driver docs.

Mongoose's `Connection#transaction()` function is a wrapper around `withTransaction()` that integrates Mongoose change tracking with transactions. For example, suppose you `save()` a document in a transaction that later fails. The changes in that document are not persisted to MongoDB. The `Connection#transaction()` function informs Mongoose change tracking that the `save()` was rolled back, and marks all fields that were changed in the transaction as modified.

```
const schema = Schema({ name: String, arr: [String], arr2: [String] });

const Test = db.model('Test', schema);

await Test.createCollection();
let doc = await Test.create({ name: 'foo', arr: ['bar'], arr2: ['foo'] });
doc = await Test.findById(doc);
await db.
  transaction(async (session) => {
    doc.arr.pull('bar');
    doc.arr2.push('bar');

    await doc.save({ session });
    doc.name = 'baz';
    throw new Error('Oops');
  }).
  catch(err => {
    assert.equal(err.message, 'Oops');
  });

const changes = doc.getChanges();
assert.equal(changes.$set.name, 'baz');
assert.deepEqual(changes.$pullAll.arr, ['bar']);
assert.deepEqual(changes.$push.arr2, { $each: ['bar'] });
assert.ok(!changes.$set.arr2);

await doc.save({ session: null });

const newDoc = await Test.findById(doc);
assert.equal(newDoc.name, 'baz');
assert.deepEqual(newDoc.arr, []);
assert.deepEqual(newDoc.arr2, ['foo', 'bar']);
```

With Mongoose Documents and `save()`

If you get a Mongoose document from `findOne()` or `find()` using a session, the document will keep a reference to the session and use that session for `save()`.

To get/set the session associated with a given document, use `doc.$session()`.

```
const User = db.model('User', new Schema({ name: String }));
const session = await db.startSession();
```

```
await session.withTransaction(async () => {
  await User.create({ name: 'foo' });

  const user = await User.findOne({ name: 'foo' }).session(session);
  // Getter/setter for the session associated with this document.
  assert.ok(user.$session());
  user.name = 'bar';
  // By default, `save()` uses the associated session
  await user.save();

  // Won't find the doc because `save()` is part of an uncommitted transaction
  const doc = await User.findOne({ name: 'bar' });
  assert.ok(!doc);
});

session.endSession();

const doc = await User.findOne({ name: 'bar' });
assert.ok(doc);
```

With the Aggregation Framework

The `Model.aggregate()` function also supports transactions. Mongoose aggregations have a `session()` helper that sets the `session` option. Below is an example of executing an aggregation within a transaction.

```
const Event = db.model('Event', new Schema({ createdAt: Date }), 'Event');
const session = await db.startSession();

await session.withTransaction(async () => {
  await Event.insertMany([
    { createdAt: new Date('2018-06-01') },
    { createdAt: new Date('2018-06-02') },
    { createdAt: new Date('2017-06-01') },
    { createdAt: new Date('2017-05-31') }
  ], { session: session });

  const res = await Event.aggregate([
    {
      $group: {
        _id: {
          month: { $month: '$createdAt' },
          year: { $year: '$createdAt' }
        },
        count: { $sum: 1 }
      }
    },
    { $sort: { count: -1, '_id.year': -1, '_id.month': -1 } }
  ]).session(session);

  assert.deepEqual(res, [
    { _id: { month: 6, year: 2018 }, count: 2 },
    { _id: { month: 6, year: 2017 }, count: 1 },
    { _id: { month: 5, year: 2017 }, count: 1 }
```

```
  ]);
});

session.endSession();
```

Advanced Usage

Advanced users who want more fine-grained control over when they commit or abort transactions can use `session.startTransaction()` to start a transaction:

```
const Customer = db.model('Customer', new Schema({ name: String }));

const session = await db.startSession();
session.startTransaction();

// This `create()` is part of the transaction because of the `session`
// option.
await Customer.create([{ name: 'Test' }], { session: session });

// Transactions execute in isolation, so unless you pass a `session`
// to `findOne()` you won't see the document until the transaction
// is committed.
let doc = await Customer.findOne({ name: 'Test' });
assert.ok(!doc);

// This `findOne()` will return the doc, because passing the `session`
// means this `findOne()` will run as part of the transaction.
doc = await Customer.findOne({ name: 'Test' }).session(session);
assert.ok(doc);

// Once the transaction is committed, the write operation becomes
// visible outside of the transaction.
await session.commitTransaction();
doc = await Customer.findOne({ name: 'Test' });
assert.ok(doc);

session.endSession();
```

You can also use `session.abortTransaction()` to abort a transaction:

```
const session = await Customer.startSession();
session.startTransaction();

await Customer.create([{ name: 'Test' }], { session: session });
await Customer.create([{ name: 'Test2' }], { session: session });

await session.abortTransaction();

const count = await Customer.countDocuments();
assert.strictEqual(count, 0);

session.endSession();
```