MongoDB ACID Transactions Documentation

Introduction

This document describes the implementation of ACID (Atomicity, Consistency, Isolation, Durability) transactions in the E-Learning Platform application. MongoDB supports multi-document transactions which ensure that data operations maintain database integrity through proper session management and error handling.

Proper Session Management

Transaction Session Lifecycle

MongoDB transactions require proper session management to ensure data integrity. Our implementation follows this pattern:

1. **Session Creation**: Create a MongoDB session
2. **Transaction Execution**: Run operations within a transaction context
3. **Commit or Rollback**: Automatically commit if successful, or rollback on error
4. **Session Cleanup**: Always end the session, even if errors occur

Implementation Details

The `transactionUtils.js` utility provides a clean abstraction for session management:

```
export const withTransaction = async (operations) => {
  // Start a session
  const session = await mongoose.startSession();

  try {
    let result;

    // Start a transaction
    await session.withTransaction(async () => {
      // Execute the operations within the transaction
      result = await operations(session);
    });

    return result;
  } catch (error) {
    console.error("Transaction failed:", error);
    throw new Error(`Transaction failed: ${error.message}`);
  } finally {
    // End the session
    await session.endSession();
  }
};
```

Key components:

- `mongoose.startSession()` : Creates a new MongoDB session
- `session.withTransaction()` : Executes operations within a transaction context
- `finally` block: Ensures session is always closed, preventing resource leaks
- Error handling: Captures and logs failures while preserving the error context

Transaction Implementation

Enrollment Operations

Enrollment operations are particularly critical as they involve establishing relationships between users and courses. Our implementation ensures:

1. **Data Integrity**: Students can't be enrolled multiple times in the same course
2. **Atomicity**: All database operations either succeed or fail as a unit
3. **Consistency**: Database remains in a valid state before and after transactions

Example from `enrollment/dao.js` :

```
export const createEnrollment = async (enrollmentData) => {
  try {
    return await withTransaction(async (session) => {
      // Check if enrollment already exists to prevent duplicates
      const existingEnrollment = await Enrollment.findOne({
        user: enrollmentData.user,
        course: enrollmentData.course
      }).session(session);

      if (existingEnrollment) {
        throw new Error("Student is already enrolled in this course");
      }

      // Create new enrollment within the transaction
      const newEnrollment = new Enrollment(enrollmentData);
      const savedEnrollment = await newEnrollment.save({ session });

      // Return the new enrollment with populated fields
      return await Enrollment.findById(savedEnrollment._id)
        .populate("user", "username firstName lastName email")
        .populate("course", "number name")
        .session(session);
    });
  } catch (error) {
    throw error;
  }
};
```

Passing the Session Object

For transactions to work correctly, the session object must be passed to all database operations within the transaction:

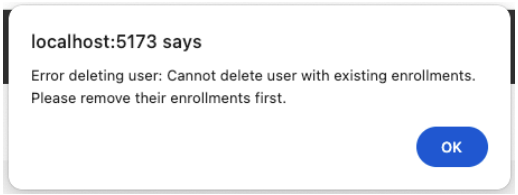
- `Model.find().session(session)`
- `Model.findOne().session(session)`
- `Model.findById().session(session)`
- `document.save({ session })`
- `Model.findByIdAndUpdate(id, data, { session })`

This ensures all operations are part of the same transaction context.

Referential Integrity in E-Learning Application

Overview

The E-Learning application enforces referential integrity between users, courses, and enrollments to maintain data consistency. When a user or course is deleted, the system checks for existing enrollments before allowing the deletion. This prevents orphaned records and ensures that all relationships are valid. When user tries to delete a course or user that has existing enrollments, the system will throw an error and prevent the deletion.



Implementation Details

Schema Design

The enrollment schema establishes relationships with both user and course documents:

```
// enrollment schema
const enrollmentSchema = new mongoose.Schema({
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "user",
    required: true
  },
  course: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "course",
    required: true
  },
  // ...other fields
});
```

Deletion Constraints

When deleting a user or course, the system first checks for existing enrollments:

```
// From user/dao.js
export const deleteUser = async (id) => {
  try {
    // Check if user has any enrollments
    const enrollments = await Enrollment.find({ user: id });
    if (enrollments.length > 0) {
      throw new Error("Cannot delete user with existing enrollments. Please remove
their enrollments first.");
    }

    // If no enrollments, proceed with deletion
    return await User.findByIdAndDelete(id);
  } catch (error) {
    throw new Error(`Error deleting user: ${error.message}`);
  }
};


// From course/dao.js
export const deleteCourse = async (id) => {
  try {
    // Check if course has any enrollments
    const enrollments = await Enrollment.find({ course: id });
    if (enrollments.length > 0) {
      throw new Error("Cannot delete course with existing enrollments. Please
remove course enrollments first.");
    }

    // If no enrollments, proceed with deletion
    return await Course.findByIdAndDelete(id);
  } catch (error) {
    throw new Error(`Error deleting course: ${error.message}`);
  }
};
```

Error Handling (Frontend)

The frontend displays meaningful error messages when a deletion violates referential integrity:

```
const handleDelete = async (id) => {
  try {
    await deleteUser(id)
    fetchUsers()
  } catch (error) {
    console.error('Error deleting user:', error)
    const errorMessage = error.response?.data?.message ||
      "Cannot delete this user. They may have active enrollments that must be
removed first."
    alert(errorMessage)
  }
}
```

ACID Properties and Design Benefits

This implementation supports the Consistency property of ACID transactions by:

1. **Preventing Orphaned Records**: No enrollment can exist without both a valid user and course
2. **Maintaining Data Integrity**: Deletion order is enforced (enrollments must be deleted first)
3. **Explicit Dependencies**: The relationship between entities is clear and enforced

Conclusion

This design pattern ensures data consistency at the application level rather than relying solely on database constraints. While it requires additional application logic to handle cascading deletes, it provides greater control over the deletion process and prevents data corruption that could occur with orphaned references.

Error Handling and Rollback Mechanisms

Automatic Rollback

MongoDB automatically rolls back all changes made within a transaction if any operation fails:

1. If any operation throws an error, the entire transaction is aborted
2. All changes made within the transaction are rolled back
3. The database returns to the state it was in before the transaction started

Custom Error Handling

Our implementation enhances MongoDB's automatic rollbacks with custom error handling:

```
try {
  // Operations within transaction
} catch (error) {
  console.error("Transaction failed:", error);
  throw new Error(`Transaction failed: ${error.message}`);
}
```

On the API side, we provide detailed error responses:

```
router.post("/api/enrollments", async (req, res) => {
  try {
    const newEnrollment = await dao.createEnrollment(req.body);
    res.status(201).json(newEnrollment);
  } catch (error) {
    console.error("Enrollment creation failed:", error);
    res.status(400).json({
      message: error.message,
      transactionFailed: true
    });
  }
});
```

Client-Side Error Handling

The client application detects transaction failures and displays appropriate messages:

```
try {
  // API call
} catch (error) {
  if (error.response?.data?.transactionFailed) {
    alert(`Transaction failed: ${error.response.data.message}`)
  } else {
    alert(`Error: ${error.response?.data?.message || error.message}`)
  }
}
```

Test Cases and Demonstrations

We've implemented demonstration cases to show both successful transactions and proper rollback behavior.

1. Successful Transaction Demonstration

The `demonstrateSuccessfulTransaction()` function shows a multi-step transaction:

```
export const demonstrateSuccessfulTransaction = async () => {
  try {
    console.log("Starting successful transaction demonstration...");

    const result = await withTransaction(async (session) => {
      // 1. Create a test user
      const testUser = new User({
        // user data...
      });
      await testUser.save({ session });

      // 2. Create a test course
      const testCourse = new Course({
        // course data...
      });
      await testCourse.save({ session });

      // 3. Create an enrollment
      const testEnrollment = new Enrollment({
        user: testUser._id,
        course: testCourse._id,
        // enrollment data...
      });
      await testEnrollment.save({ session });

      return {
        user: testUser,
        course: testCourse,
        enrollment: testEnrollment
      };
    });

    console.log("Transaction successful!", result);
    return result;
  } catch (error) {
    console.error("Transaction demonstration failed:", error);
    throw error;
  }
};
```

This creates three related documents in a single atomic transaction.

2. Rollback Demonstration

The `demonstrateFailedTransaction()` function shows proper rollback behavior:

```
export const demonstrateFailedTransaction = async () => {
  try {
    console.log("Starting failed transaction demonstration...");

    await withTransaction(async (session) => {
      // 1. Create a test user
      const testUser = new User({
        // user data...
      });
      await testUser.save({ session });

      // 2. Create a test course
      const testCourse = new Course({
        // course data...
      });
      await testCourse.save({ session });

      // 3. Create an enrollment with invalid data to cause failure
      const testEnrollment = new Enrollment({
        user: testUser._id,
        course: testCourse._id,
        // Invalid status to trigger validation error
        status: "INVALID_STATUS"
      });
      await testEnrollment.save({ session });

      return "This should not be returned due to rollback";
    });
  } catch (error) {
    console.log("Transaction failed as expected with rollback:", error.message);

    // Verify rollback by checking if the user and course were not saved
    const user = await User.findOne({ username: "rollback_test_user" });
    const course = await Course.findOne({ number: "TRX102" });

    console.log("User was rolled back:", user === null);
    console.log("Course was rolled back:", course === null);

    return {
      rolledBack: true,
      userExists: user !== null,
      courseExists: course !== null
    };
  }
};
```

This demonstrates:

1. Creating multiple documents in a single transaction
2. Intentionally causing a validation error
3. Verifying all operations are rolled back (no partial commits)
4. Checking that none of the documents were persisted to the database

3. Complex Transaction Example: Bulk Enrollment

The `bulkEnrollStudents` function demonstrates a more complex transaction that can handle partial failures within a transaction:

```
export const bulkEnrollStudents = async (courseId, userIds) => {
  try {
    return await withTransaction(async (session) => {
      const results = {
        successful: [],
        failed: []
      };

      for (const userId of userIds) {
        try {
          // Check for existing enrollment
          // Create enrollment if needed
          // Record success
        } catch (error) {
          // Record individual failure without failing entire transaction
          results.failed.push({
            userId,
            reason: error.message
          });
        }
      }

      // Only roll back if nothing succeeded
      if (results.successful.length === 0 && userIds.length > 0) {
        throw new Error("All enrollments failed");
      }

      return results;
    });
  } catch (error) {
    throw error;
  }
};
```

This implementation:

1. Processes multiple enrollments within a single transaction
2. Tracks individual successes and failures
3. Allows partial success (some enrollments succeed while others fail)
4. Only triggers a complete rollback if all enrollments fail

Running the Demonstrations

You can execute both transaction demonstrations by running:

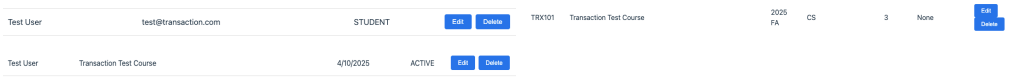
```
node elearning/utils/transactionDemo.js
```

This is the log information in the console after running the demonstration:

```
MONGODB_URI available: Yes
Connecting to MongoDB Atlas...
Connected to MongoDB Atlas
=== TRANSACTION DEMONSTRATIONS ===

1. SUCCESSFUL TRANSACTION DEMO:
Starting successful transaction demonstration...
Transaction successful! {
  user: {
    username: 'test_transaction_user',
    password: 'password123',
    firstName: 'Test',
    email: 'test@transaction.com',
    lastName: 'User',
    userID: 99999,
    role: 'STUDENT',
    _id: new ObjectId('67f834bd4fbb8418da4dccbe')
  },
  course: {
    number: 'TRX101',
    name: 'Transaction Test Course',
    term: '2025 FA',
    department: 'CS',
    credits: 3,
    _id: new ObjectId('67f834be4fbb8418da4dccc1'),
    lessons: [],
    __v: 0
  },
  enrollment: {
    user: new ObjectId('67f834bd4fbb8418da4dccbe'),
    course: new ObjectId('67f834be4fbb8418da4dccc1'),
    enrollmentDate: 2025-04-10T21:14:38.091Z,
    status: 'ACTIVE',
    _id: new ObjectId('67f834be4fbb8418da4dccc3'),
    __v: 0
  }
}

2. FAILED TRANSACTION WITH ROLLBACK DEMO:
Starting failed transaction demonstration...
Transaction failed as expected with rollback: Transaction failed: enrollment validation failed:
status: `INVALID_STATUS` is not a valid enum value for path `status`.
User was rolled back: true
Course was rolled back: true
Failed transaction result: { rolledBack: true, userExists: false, courseExists: false }

3. COMPLEX TRANSACTION WITH PARTIAL SUCCESS DEMO:
Starting bulk enrollment demonstration with 5 valid users and 1 invalid user...
Attempting to enroll 6 users (5 valid, 1 invalid)
Bulk enrollment results: {
  successful: 5,
  failed: 1,
  details: {
    successful: [ [Object], [Object], [Object], [Object], [Object] ],
    failed: [ [Object] ]
  }
```

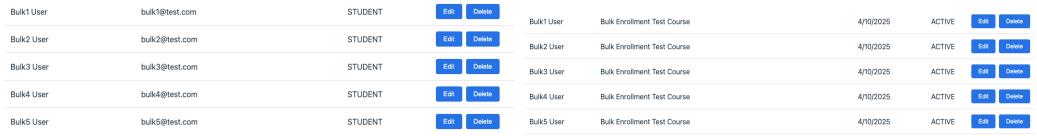
```
}
Bulk enrollment result summary: { successful: 5, failed: 1 }
```

- The first function successfully creates a user, course, and enrollment.



- The second function demonstrates a failed transaction with proper rollback, confirming that the user and course were not persisted. No documents were created in the database.

- The third function demonstrates a sophisticated transaction pattern where multiple operations are processed within a single transaction context. It successfully creates and enrolls 5 valid student users in the course, while properly handling the failure of 1 invalid user without rolling back the entire transaction. This illustrates how our transaction implementation can maintain data consistency while gracefully managing partial failures in bulk operations.



The output demonstrates:

1. **Successful Transaction**: Creates a user, course, and enrollment in a single atomic operation, with all three objects persisted to the database.

2. **Failed Transaction with Rollback**: Attempts to create objects with invalid data (invalid enrollment status), causing a validation error. The output confirms that both the user and course objects were properly rolled back and not persisted to the database, maintaining data integrity.

3. **Complex Transaction with Partial Success**: Demonstrates a bulk enrollment operation where multiple enrollments are processed in a single transaction. The transaction successfully enrolls 5 users while properly handling 1 invalid enrollment without rolling back the entire transaction.

These demonstrations verify that our transaction implementation correctly handles both successful operations and proper rollbacks when errors occur.

MongoDB vs. RDBMS Transactions

Key Differences

Feature	MongoDB	Traditional RDBMS (e.g., MySQL)
Transaction Scope	Multi-document transactions across collections	Multi-table transactions
Performance Impact	Higher overhead, especially across shards	Optimized for transactional workloads
Isolation Levels	Limited options (read concern/write concern)	Multiple isolation levels (READ COMMITTED, REPEATABLE READ, etc.)
Locking Model	Document-level locking	Row-level or table-level locking
Implementation	Explicit session management	Often managed by connection pooling

MongoDB Transaction Limitations

1. **Performance**: Transactions in MongoDB have higher overhead compared to single document operations
2. **Distributed Transactions**: Transactions across sharded collections have additional complexity and constraints
3. **Duration**: Long-running transactions should be avoided
4. **Session Management**: Sessions must be explicitly created, passed, and closed

When to Use MongoDB Transactions

MongoDB transactions are ideal for:

1. Operations that must guarantee atomicity across multiple documents
2. Maintaining data integrity when modifying related data
3. Scenarios where partial updates would lead to inconsistent state

For our e-learning platform, transactions are most appropriate for:

- Enrollment operations (linking users and courses)
- User registration processes that span multiple collections
- Data operations that must maintain referential integrity

Best Practices

1. **Keep transactions short**: Long-running transactions can impact performance
2. **Limit the scope**: Include only necessary operations in transactions
3. **Error handling**: Always catch errors and handle rollbacks appropriately
4. **Session management**: Always close sessions in a finally block
5. **Testing**: Test both successful cases and failure scenarios

Conclusion

MongoDB's ACID transaction support enables us to maintain data integrity in our e-learning platform. By following proper session management practices and implementing appropriate error handling, we can ensure that critical operations like enrollment maintain database consistency even when errors occur.

The demonstration code provides clear examples of both successful transactions and proper rollback behavior, helping developers understand how transactions work in practice.