Step 5: Database Abstraction Layer

ORDB - Universal Database Interface

What Changed?

- Step 3: MongoDB with native driver
- Step 4: SQLite with native driver
- Step 5: **ORDB abstraction layer**
  - Switch databases with **one configuration change**
  - Same application code for both databases
  - **Design pattern:** Adapter + Repository patterns

The Problem

Before Step 5:

```
// MongoDB code
const posts = await db.collection('posts').find({}).toArray();

// SQLite code
db.all('SELECT * FROM posts', [], (err, posts) => { ... });
```

Problem: Different API for each database

- Hard to switch databases
- Code duplication
- Database vendor lock-in

The Solution: ORDB

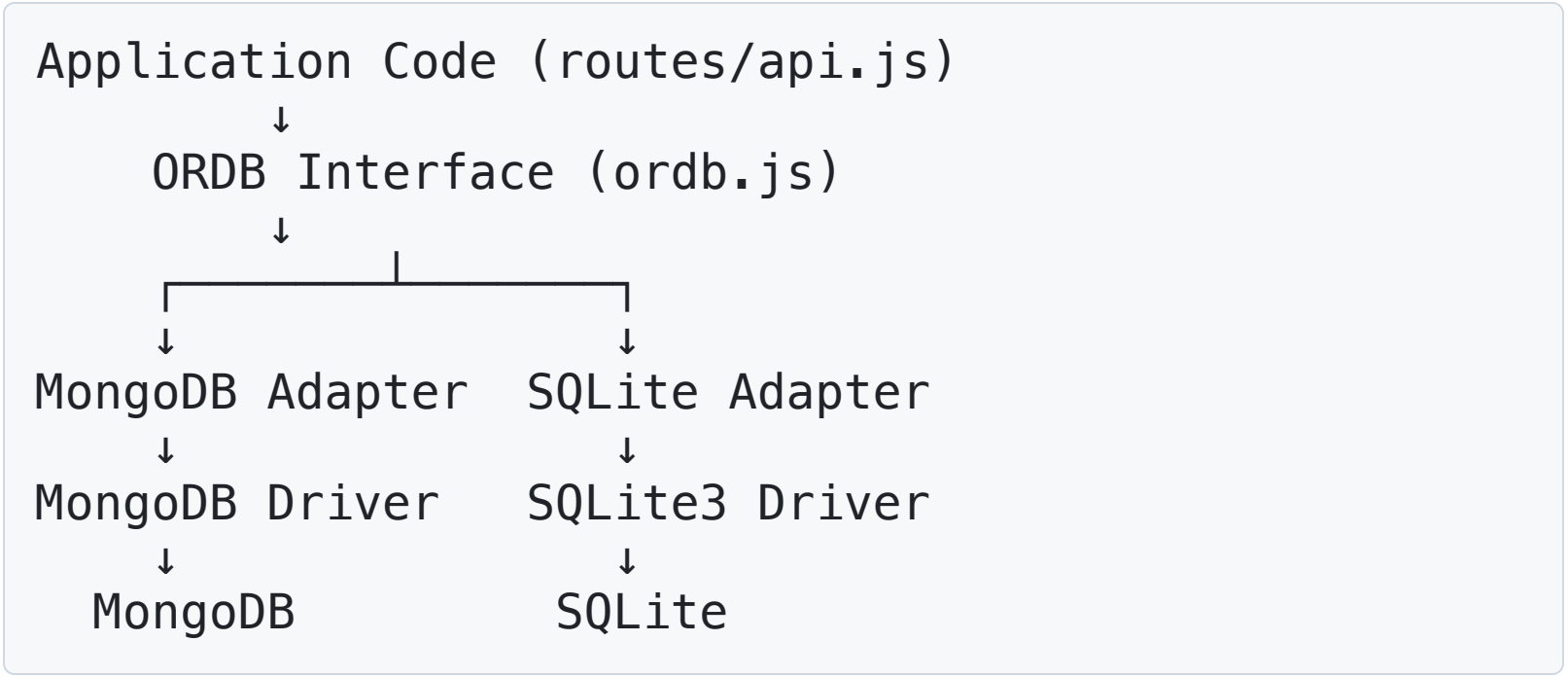
Object-Relational Database Abstraction Layer

```
// Works with BOTH MongoDB and SQLite!
const posts = await db.findAll('posts', {}, { sort: { _id: 1 } });
```

Benefits:

- **Single interface** for all databases
- **Easy switching** - change one configuration file
- **No code changes** in routes or business logic

Architecture Overview



Separation of concerns: App doesn't know which database

Directory Structure

```
.
├── .env                          # Configuration file (NEW!)
├── .env.example                  # Template
├── db/
│   ├── ordb.js                   # Abstract base class
│   ├── mongodb-adapter.js        # MongoDB implementation
│   ├── sqlite-adapter.js         # SQLite implementation
│   ├── bridge.js                 # Auto-selects adapter
│   └── uri.js                    # Loads .env
├── routes/
│   ├── api.js                    # Database-agnostic
│   └── router.js
└── index.js
```

Environment Configuration

Simplified from previous steps:

`db/uri.js` :

```
import dotenv from "dotenv";

// Load .env from current directory (not parent!)
dotenv.config();  // Looks for .env in project root

const user = process.env.MONGO_USER;
const password = process.env.MONGO_PASSWORD;
// ... rest of MongoDB setup
```

Key change: `.env` in **project root**, not parent directory

Why This Change?

Previous steps (Step 1-4):

```
dotenv.config({ path: "../.env" });  // Parent directory
```

Step 5:

```
dotenv.config();  // Current directory (default)
```

Reason: Simpler and standard

- Each project has its own `.env`
- No confusion about file location
- Follows Node.js conventions

ORDB Interface (Abstract Class)

```
export class ORDB {
  async connect() { throw new Error('Must implement'); }

  async findAll(collection, filter = {}, options = {}) {
    throw new Error('Must implement');
  }

  async findOne(collection, filter) {
    throw new Error('Must implement');
  }

  async insertOne(collection, data) {
    throw new Error('Must implement');
  }

  async updateOne(collection, filter, update) {
    throw new Error('Must implement');
  }

  async deleteOne(collection, filter) {
    throw new Error('Must implement');
  }
}
```

Why Abstract Class?

Enforces consistency:

- All adapters must implement same methods
- Same method signatures
- Same behavior

Design Pattern:

- **Repository Pattern** - Abstracts data access
- **Adapter Pattern** - Converts one interface to another

MongoDB Adapter

```
export class MongoDBAdapter extends ORDB {
  async findAll(collection, filter = {}, options = {}) {
    const coll = this.db.collection(collection);
    let query = coll.find(filter);

    if (options.sort) {
      query = query.sort(options.sort);
    }

    return await query.toArray();
  }

  async insertOne(collection, data) {
    if (!data._id) {
      data._id = await this.getNextId(collection);
    }
    await this.db.collection(collection).insertOne(data);
    return data;
  }
}
```

SQLite Adapter

```
export class SQLiteAdapter extends ORDB {
  async findAll(collection, filter = {}, options = {}) {
    return new Promise((resolve, reject) => {
      let sql = `SELECT * FROM ${collection}`;
      // Build WHERE clause from filter
      // Add ORDER BY from options.sort

      this.db.all(sql, params, (err, rows) => {
        if (err) reject(err);
        else resolve(rows || []);
      });
    });
  }

  async insertOne(collection, data) {
    // INSERT SQL with AUTOINCREMENT
    // Return data with this.lastID
  }
}
```

Key Differences Hidden by Adapters

MongoDB → SQLite Translation

MongoDB	SQLite
Collections	Tables
`find({})`	`SELECT *`
`{ _id: 5 }` filter	`WHERE _id = 5`
`{ _id: 1 }` sort	`ORDER BY _id ASC`
Callbacks/Promises	Promises only

Adapters handle all conversions!

Bridge: Auto-Selecting Adapter

```
import uri from './uri.js'  // Loads .env

function createDatabaseAdapter() {
  const dbType = process.env.DB_TYPE || 'sqlite';

  switch (dbType.toLowerCase()) {
    case 'mongodb':
      return new MongoDBAdapter(uri, databasename);

    case 'sqlite':
      return new SQLiteAdapter(dbPath);

    default:
      return new SQLiteAdapter(dbPath);
  }
}

export const db = createDatabaseAdapter();
```

Configuration File (.env)

In project root directory:

```
# Database Configuration
DB_TYPE=sqlite

# MongoDB Configuration (if DB_TYPE=mongodb)
MONGO_USER=your_username
MONGO_PASSWORD=your_password
MONGO_CLUSTER=cluster0.xxxxx.mongodb.net
MONGO_DATABASE=todoapp

# SQLite Configuration (if DB_TYPE=sqlite)
SQLITE_PATH=./todoapp.sqlite
```

Single file for all configuration!

Switching Databases

Step 1: Edit `.env` file in project root

```
# For SQLite (default)
DB_TYPE=sqlite

# For MongoDB
DB_TYPE=mongodb
```

Step 2: Restart server

```
npm start
```

That's it! No code changes needed

Application Code - Before

Step 3 (MongoDB):

```
router.get('/posts', async (req, res) => {
  const posts = await db.collection('posts')
    .find({})
    .sort({ _id: 1 })
    .toArray();
  res.json(posts);
});
```

Step 4 (SQLite):

```
router.get('/posts', (req, res) => {
  const sql = 'SELECT * FROM posts ORDER BY _id ASC';
  db.all(sql, [], (err, posts) => {
    res.json(posts);
  });
});
```

Application Code - After (Step 5)

Works with BOTH databases:

```
router.get('/posts', async (req, res) => {
  try {
    const posts = await db.findAll('posts', {}, {
      sort: { _id: 1 }
    });
    res.json(posts);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch posts' });
  }
});
```

Same code, any database!

CRUD: Create - Unified

```
// POST /api/posts
router.post('/posts', async (req, res) => {
  try {
    const { title, date } = req.body || {};

    if (!title) {
      return res.status(400).json({ error: 'title is required' });
    }

    // Works with both MongoDB and SQLite!
    const newPost = await db.insertOne('posts', {
      title,
      date: date || ''
    });

    res.status(201).json(newPost);
  } catch (error) {
    res.status(500).json({ error: 'Failed to create post' });
  }
});
```

CRUD: Read - Unified

```
// GET /api/posts/:id
router.get('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);

    // Works with both databases!
    const post = await db.findOne('posts', { _id: id });

    if (!post) {
      return res.status(404).json({ error: 'Post not found' });
    }

    res.json(post);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch post' });
  }
});
```

CRUD: Update - Unified

```
// PUT /api/posts/:id
router.put('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);
    const { title, date } = req.body || {};

    const update = {};
    if (title !== undefined) update.title = title;
    if (date !== undefined) update.date = date;

    // Works with both databases!
    const updatedPost = await db.updateOne(
      'posts',
      { _id: id },
      update
    );

    res.json(updatedPost);
  } catch (error) {
    res.status(500).json({ error: 'Failed to update post' });
  }
});
```

CRUD: Delete - Unified

```
// DELETE /api/posts/:id
router.delete('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);

    // Works with both databases!
    const deleted = await db.deleteOne('posts', { _id: id });

    if (!deleted) {
      return res.status(404).json({ error: 'Post not found' });
    }

    res.json({ ok: true, deletedId: id });
  } catch (error) {
    res.status(500).json({ error: 'Failed to delete post' });
  }
});
```

The Async Pattern Problem

Remember Step 4?

Database	Native API
MongoDB	Promise-based (modern)
SQLite	Callback-based (old style)

Challenge: How to make a **unified interface**?

Solution: Adapters convert everything to Promises!

MongoDB Adapter: Already Promises

```
export class MongoDBAdapter extends ORDB {
  // MongoDB native driver already returns Promises
  async findOne(collection, filter) {
    // This is already a Promise!
    return await this.db.collection(collection).findOne(filter);
  }
}
```

Easy: MongoDB driver is already Promise-based

- Just `await` the result
- Return directly

SQLite Adapter: Wrapping Callbacks

```
export class SQLiteAdapter extends ORDB {
  async findOne(collection, filter) {
    // Create a NEW Promise object
    return new Promise((resolve, reject) => {
      const sql = 'SELECT * FROM posts WHERE _id = ?';

      // Start SQLite query (non-blocking)
      this.db.get(sql, [filter._id], (err, row) => {
        // This callback runs LATER when query completes
        if (err) reject(err);
        else resolve(row || null);
      });
      // Function continues immediately (doesn't wait here)
    });
    // Returns the Promise object (still pending)
  }
}
```

Key Concept: Promise Execution

Important: Promise does NOT wait for callback!

```
return new Promise((resolve, reject) => {
  // 1. Promise executor runs IMMEDIATELY
  this.db.get(sql, params, (err, row) => {
    // 2. Callback scheduled for LATER
    if (err) reject(err);
    else resolve(row);
  });
  // 3. Execution continues HERE (immediately)
});
// 4. Returns Promise object (still pending)
```

Flow:

1. Promise created
2. SQLite query started (asynchronous)
3. Promise returned immediately (pending state)
4. Callback executes later
5. Callback calls resolve/reject (settles Promise)

Execution Timeline

```
const post = await db.findOne('posts', { _id: 5 });
```

Timeline:

```
Time 0ms:   findOne() called
          ↓  new Promise() created
          ↓  db.get() started (asynchronous I/O)
          ↓  Promise returned (pending)
          ↓  await starts waiting

Time 5ms:   SQLite query running in background...

Time 10ms:  Query complete!
          ↓  Callback executes
          ↓  resolve(row) called
          ↓  Promise settled (fulfilled)
          ↓  await returns the value

Time 11ms:  post variable has the data
```

Promise States

```
const promise = new Promise((resolve, reject) => {
  this.db.get(sql, params, (err, row) => {
    if (err) reject(err);
    else resolve(row);
  });
});
```

Three states:

1. **Pending** - Created, waiting for callback
2. **Fulfilled** - resolve() called, has value
3. **Rejected** - reject() called, has error

await waits for state change: Pending → Fulfilled/Rejected

Why This Works

Key insight:

Promise doesn't **execute** the callback.

Promise **captures** the callback's result through `resolve/reject`.

```
// Promise gives callback two functions:
new Promise((resolve, reject) => {
  // Callback can call these anytime:
  resolve(value);  // Success → await returns value
  reject(error);   // Error → await throws error
});
```

Bridge pattern: Connects callback world to Promise world

Complete Example with Comments

```
async findOne(collection, filter) {
  // Return a new Promise (immediately)
  return new Promise((resolve, reject) => {
    const sql = 'SELECT * FROM posts WHERE _id = ?';

    // Start async operation (returns immediately)
    this.db.get(sql, [filter._id], (err, row) => {
      // Callback executes LATER (when query done)
      if (err) {
        reject(err);  // Tell Promise: "failed"
      } else {
        resolve(row || null);  // Tell Promise: "success with data"
      }
    });
    // Execution continues here immediately
    // Promise is returned in "pending" state
  });
  // await will pause here until resolve/reject called
}
```

Using the Unified Interface

Application code (works for BOTH):

```
// This works whether it's MongoDB or SQLite!
const post = await db.findOne('posts', { _id: 5 });
```

Behind the scenes:

- **MongoDB**: Returns Promise directly
- **SQLite**: Wraps callback in Promise

Application doesn't know or care!

Design Patterns Used

1. Abstract Factory Pattern

- `bridge.js` creates appropriate adapter

2. Adapter Pattern

- Converts database-specific APIs to unified interface

3. Repository Pattern

- Abstracts data access layer

4. Promise Wrapper Pattern

- Bridges callback-based APIs to Promise-based

Benefits of Abstraction Layer

1. **Database Independence** - Switch anytime
2. **Testability** - Mock database easily
3. **Maintainability** - Changes in one place
4. **Scalability** - Add new databases without changing app
5. **Clean Code** - Separation of concerns
6. **Unified Async** - All methods use async/await

Real-World Applications

When to use abstraction layers:

- Multi-tenant applications (different DBs per customer)
- Cloud migration (on-premise → cloud)
- Development/production environments (SQLite dev, MongoDB prod)
- Database performance testing
- Future-proofing applications

Adding a New Database

Example: PostgreSQL

```
// 1. Create postgres-adapter.js
export class PostgreSQLAdapter extends ORDB {
  async findAll(collection, filter, options) {
    // Wrap pg callback in Promise
    return new Promise((resolve, reject) => {
      // pg library code here
    });
  }
}

// 2. Update bridge.js
case 'postgresql':
  return new PostgreSQLAdapter(config);

// 3. Update .env
DB_TYPE=postgresql
```

Trade-offs

Advantages ✅

- Database flexibility
- Cleaner code
- Easier testing
- Better maintainability

Disadvantages ❌

- Extra abstraction layer
- May not support all database-specific features
- Small performance overhead
- More complex architecture

When NOT to Use Abstraction

Don't abstract if:

- Only using one database forever
- Need database-specific features (MongoDB aggregation, PostgreSQL JSON)
- Performance is absolutely critical
- Small prototype or learning project

Abstract when:

- Multiple databases possible
- Team needs flexibility
- Long-term maintainability important

Running Application

Step 1: Copy .env.example to .env

```
cp env .env
```

Step 2: Edit .env to choose database

```
# In project root: .env
DB_TYPE=sqlite    # or mongodb
```

When using MongoDB, fill in credentials.

Step 3: Start server

```
npm install
npm start
```

Database is automatically selected!

Key Concepts

- **Abstraction Layer** - Hide implementation details
- **Adapter Pattern** - Convert one interface to another
- **Repository Pattern** - Separate data access logic
- **Promise Wrapper** - Bridge callbacks to Promises
- **Async Execution** - Promise captures callback result
- **Database Portability** - Switch databases easily

Summary

- **ORDB** provides unified database interface
- **Adapters** convert database-specific APIs
- **Promise wrapping** bridges callbacks to async/await
- **Promise executes immediately**, callback runs later
- **resolve/reject** connect callback to Promise
- **Application code** is database-agnostic
- Switch databases by **editing one line in .env**