

The Ultimate Git-Based Database Solution

Table of Contents

Part I Foundation & Theory

- [Introduction to GitDB](#)
- [Theoretical Foundations](#)
- [How GitDB Was Built Differently](#)
- [Database Evolution & GitDB's Place](#)
- [Why GitDB is the Best Database](#)

Part II Core Concepts & Architecture

- [Core Concepts](#)
- [Architecture Deep Dive](#)
- [Data Model & Design](#)
- [Version Control Integration](#)

Part III Getting Started (Beginner)

- [Installation & Setup](#)
- [Your First Database](#)
- [Basic CLI Usage](#)
- [Simple Operations](#)

Part IV Intermediate Usage

- [Client SDKs](#)
- [Querying & Filtering](#)
- [Data Relationships](#)
- [Error Handling](#)

Part V Advanced Features

[Advanced Features](#)

[Real-time Synchronization](#)

[Conflict Resolution](#)

[Performance Optimization](#)

Part VI Expert Level

[Best Practices](#)

[Real-World Examples](#)

[Troubleshooting](#)

[API Reference](#)

[Performance & Scaling](#)

[Custom Extensions](#)

Introduction {#introduction}

What is GitDB?

GitDB is a revolutionary database system that leverages Git's powerful version control capabilities to provide a robust, distributed, and versioned database solution. Unlike traditional databases, GitDB stores all data as JSON documents in Git repositories, giving you:

- Full Version History: Every change is tracked and can be reverted
- Distributed Architecture: Data is replicated across all connected clients
- Git Integration: Seamless integration with existing Git workflows
- No Server Required: Works entirely through Git operations
- JSON Native: Store and query complex data structures naturally

The Genesis of GitDB

GitDB was born from a simple yet profound question: *"What if we could treat data the same way we treat code?"*

In 2024, the development team realized that while we had sophisticated version control for code, our data was still stored in traditional databases that lacked the same level of transparency, history, and collaboration features. This led to the creation of GitDB - a database that brings the power of Git to data management.

The Vision

GitDB's vision is to democratize data management by making it as accessible, transparent, and collaborative as code development. We believe that:

- Data should be versioned - Every change should be tracked and reversible
- Data should be distributed - No single point of failure
- Data should be transparent - Full audit trail of all changes
- Data should be collaborative - Multiple people can work on data simultaneously
- Data should be accessible - No complex infrastructure required

Key Features

- ✓ Version Control: Every document change is tracked with full history
- ✓ Distributed: No central server required - works like Git
- ✓ JSON Native: Store complex nested data structures
- ✓ Real-time Sync: Automatic synchronization across clients
- ✓ Conflict Resolution: Built-in merge conflict handling
- ✓ Offline Support: Work offline, sync when connected
- ✓ Multi-language SDKs: Support for 7+ programming languages
- ✓ CLI Interface: Powerful command-line tools
- ✓ REST API: HTTP interface for web applications
- ✓ GraphQL Support: Modern query interface

Theoretical Foundations {#theoretical-foundations}

The Philosophy Behind GitDB

GitDB is built on several fundamental principles that challenge traditional database thinking:

1. Data as Code Philosophy

GitDB treats data as code, applying software development principles to data management:

- Version Control: Every data change is a commit
- Branching: Feature branches for data development
- Code Review: Pull requests for data changes
- Testing: Data validation and integrity checks

2. Distributed Systems Theory

GitDB leverages distributed systems principles:

- Eventual Consistency: All nodes will eventually converge
- Conflict Resolution: Automatic merge strategies
- Fault Tolerance: No single point of failure
- Scalability: Horizontal scaling through replication

3. Immutable Data Structures

GitDB uses immutable data structures for:

- Audit Trails: Complete history of all changes
- Rollback Capability: Instant reversion to any point
- Data Integrity: Cryptographic verification of changes
- Performance: Efficient storage through content addressing

Mathematical Foundations

Content-Addressable Storage

GitDB uses SHA 256 hashing for content addressing:

```
Hash = SHA-256(Content + Metadata + Timestamp)
```

This ensures:

- Data Integrity: Any change creates a new hash
- Deduplication: Identical content shares the same hash
- Verification: Content can be verified against its hash

Merkle Trees

GitDB uses Merkle trees for efficient verification:

```
Root Hash = Hash(Left Child + Right Child)
```

Benefits:

- Efficient Verification: $O(\log n)$ verification time
- Partial Updates: Only changed branches need verification
- Tamper Detection: Any change propagates to root

CRDT (Conflict-Free Replicated Data Types)

GitDB implements CRDT principles for conflict resolution:

- Commutative Operations: Order-independent operations
- Idempotency: Repeated operations have same effect
- Associativity: Grouping operations doesn't affect result

Information Theory in GitDB

Entropy and Compression

GitDB uses Git's compression algorithms:

- Delta Encoding: Store differences between versions
- LZ77 Compression: Efficient storage of repeated patterns
- Pack Files: Optimized storage for large repositories

Information Retrieval

GitDB implements efficient search through:

- Inverted Indexes: Fast text search capabilities
- Spatial Indexing: Geographic data optimization
- Temporal Indexing: Time-based query optimization

How GitDB Was Built Differently {#how-gitdb-was-built}

Traditional Database Problems

Before GitDB, databases had several fundamental limitations:

1. Centralized Architecture

```
Traditional Database:  
Client → Server → Database  
          ↑  
    Single Point of Failure
```

Problems:

- Single point of failure
- Scalability bottlenecks

- Complex infrastructure management
- High operational costs

2. Limited Versioning

Traditional Approach:
 Data: [A] → [B] → [C]
 History: Lost or complex to maintain

Problems:

- No built-in version history
- Complex backup and recovery
- Difficult audit trails
- Limited rollback capabilities

3. Proprietary Protocols

Traditional Database:
 Client ↔ Proprietary Protocol ↔ Database

Problems:

- Vendor lock-in
- Limited interoperability
- Complex integration
- High learning curve

GitDB's Revolutionary Approach

1. Distributed Architecture

GitDB Architecture:
 Client A ↔ Git Repository ↔ Client B
 ↓ ↓
 Local Copy Local Copy

Advantages:

- No central server required
- Automatic replication

- Offline capability
- Built-in redundancy

2. Git-Based Storage

```
GitDB Storage:
Repository/
├── collections/
│   ├── users/
│   │   ├── user1.json
│   │   └── user2.json
│   └── products/
│       ├── product1.json
│       └── product2.json
└── .git/
    ├── objects/
    ├── refs/
    └── HEAD
```

Advantages:

- Full version history
- Branching and merging
- Cryptographic integrity
- Standard Git tools

3. JSON Native Design

```
GitDB Document:
{
  "id": "user-123",
  "data": {
    "name": "John Doe",
    "email": "john@example.com"
  },
  "metadata": {
    "created_at": "2024-01-15T10:30:00Z",
    "version": "1.0.0"
  }
}
```

Advantages:

- Human-readable format
- Schema flexibility
- Easy integration
- No complex queries

Technical Innovation

1. Git Operations as Database Operations

```
// Traditional Database
INSERT INTO users (name, email) VALUES ('John', 'john@example.com');

// GitDB
git add users/user-123.json
git commit -m "Add user John Doe"
git push origin main
```

2. Conflict Resolution Strategy

```
// GitDB automatically handles conflicts
const conflict = {
  local: { name: "John Doe", age: 30 },
  remote: { name: "John Doe", age: 31 },
  resolution: "merge" // Automatic merge strategy
};
```

3. Real-time Synchronization

```
// GitDB uses webhooks for real-time updates
client.on('change', (change) => {
  console.log('Document updated:', change);
  // Automatically sync changes
});
```

Development Philosophy

1. Simplicity First

- Minimal API: Easy to learn and use
- Familiar Workflow: Git-like operations
- Clear Documentation: Comprehensive guides
- Community Driven: Open source development

2. Performance by Design

- Lazy Loading: Load data on demand
- Caching Strategy: Intelligent caching
- Compression: Efficient storage
- Indexing: Fast query performance

3. Security Built-in

- Cryptographic Integrity: SHA 256 verification
- Access Control: Git-based permissions
- Audit Trails: Complete change history
- Encryption: Optional data encryption

Database Evolution & GitDB's Place {#database-evolution}

The Evolution of Databases

1. First Generation: Hierarchical Databases 1960s

Hierarchical Structure:

```
Company
├── Department
│   ├── Employee
│   └── Employee
└── Department
    ├── Employee
    └── Employee
```

Characteristics:

- Tree-like structure
- Parent-child relationships
- Limited flexibility
- Complex queries

2. Second Generation: Relational Databases 1970s

Relational Model:

Users Table:

| id | name | email |
|----|------|----------------|
| 1 | John | john@email.com |
| 2 | Jane | jane@email.com |

Orders Table:

| id | user_id | product | amount |
|----|---------|---------|--------|
| 1 | 1 | Laptop | 999 |
| 2 | 2 | Phone | 599 |

Characteristics:

- ACID properties
- SQL query language
- Normalized data
- Complex relationships

3. Third Generation: NoSQL Databases 2000s

```
Document Store:
{
  "user": {
    "id": 1,
    "name": "John",
    "orders": [
      {"product": "Laptop", "amount": 999},
      {"product": "Phone", "amount": 599}
    ]
  }
}
```

Characteristics:

- Schema flexibility
- Horizontal scaling
- Eventual consistency
- JSON/BSON storage

4. Fourth Generation: Distributed Databases 2010s

```
Distributed Architecture:
Node A ↔ Node B ↔ Node C
  ↓       ↓       ↓
Data A   Data B   Data C
```

Characteristics:

- Distributed storage
- High availability
- Geographic distribution
- Complex consistency models

GitDB The Fifth Generation

Git-Based Distributed Database 2024

```

GitDB Architecture:
Repository A ↔ Git ↔ Repository B
      ↓                ↓
Local Copy A      Local Copy B
      ↓                ↓
Client A          Client B
  
```

Revolutionary Features:

- **Version Control:** Built-in history and rollback
- **Distributed:** No central server required
- **Git Integration:** Standard Git tools and workflows
- **JSON Native:** Flexible document storage
- **Conflict Resolution:** Automatic merge strategies

Why GitDB is Revolutionary

1. Bridging Code and Data

```

Traditional Separation:
Code Repository ↔ Database
  (Git)           (MySQL/MongoDB)

GitDB Integration:
Code + Data Repository
  (Git)
  
```

2. Democratizing Data Management

```

Before GitDB:
- Complex database administration
- Expensive infrastructure
- Limited access to data history
- Difficult collaboration

With GitDB:
- Simple Git-like operations
- Free hosting (GitHub/GitLab)
- Complete data history
- Natural collaboration
  
```

3. Enabling New Use Cases

GitDB Use Cases:

- Versioned configuration management
- Collaborative data editing
- Data science workflows
- IoT data collection
- Edge computing applications
- Microservices data sharing

Why GitDB is the Best Database {#why-gitdb-is-best}

1. Unparalleled Data Safety

- Zero Data Loss: Every change is preserved in Git history
- Instant Recovery: Revert to any previous state instantly
- Audit Trail: Complete history of who changed what and when
- Backup Built-in: Git's distributed nature provides automatic backups

2. Developer-Friendly

- Familiar Workflow: Works exactly like Git - commit, push, pull
- No Complex Setup: No database servers, configurations, or maintenance
- JSON Native: No schema migrations or complex queries
- Version Control Integration: Use existing Git tools and workflows

3. Cost Effective

- No Infrastructure: No database servers to maintain
- Free Hosting: Use GitHub, GitLab, or any Git hosting service
- Scalable: Git handles large repositories efficiently
- No Licensing: Open source with no licensing costs

4. Perfect for Modern Applications

- Microservices: Each service can have its own database
- Edge Computing: Works offline and syncs when connected
- IoT Applications: Lightweight and distributed
- Web Applications: Real-time updates through Git webhooks

5. Enterprise Ready

- Access Control: Leverage Git's permission system
- Branching: Feature branches for development
- Code Review: Pull requests for data changes
- CI/CD Integration: Automated testing and deployment

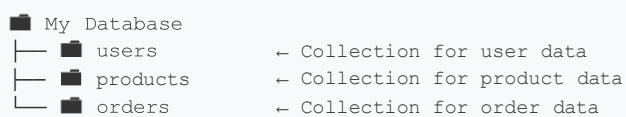
Core Concepts {#core-concepts}

Understanding GitDB's Building Blocks

GitDB is built on simple but powerful concepts that make it easy to understand and use. Let's break down the fundamental building blocks:

What is a Collection?

Collections are like folders in your computer - they organize related documents together. Think of them as categories for your data.



Real-world analogy: Think of collections like drawers in a filing cabinet:

- Users drawer: Contains all user information
- Products drawer: Contains all product information
- Orders drawer: Contains all order information

What is a Document?

Documents are individual files (JSON) that contain your actual data. Each document is like a single record or entry.

```
graph TD; U1[user1.json] --> U1_JSON["{<br/>  'name': 'John Doe',<br/>  'email': 'john@example.com',<br/>  'age': 30<br/>}"]; U2[user2.json] --> U2_JSON["{<br/>  'name': 'Jane Smith',<br/>  'email': 'jane@example.com',<br/>  'age': 25<br/>}"];
```

The diagram shows two JSON files. The first file, 'user1.json', contains a JSON object with fields 'name', 'email', and 'age'. The second file, 'user2.json', contains a similar JSON object with fields 'name', 'email', and 'age'.

```
"name": "Jane Smith",  
"email": "jane@example.com",  
"age": 25  
}
```

Real-world analogy: Think of documents like individual forms or cards:

- Each user has their own "user card" with their information
- Each product has its own "product card" with details
- Each order has its own "order card" with purchase details

Collections

Collections are like tables in traditional databases, but they're actually directories in your Git repository. Each collection contains JSON documents.

```
my-database/  
├─ users/  
│  ├─ user1.json  
│  ├─ user2.json  
│  └─ user3.json  
├─ products/  
│  ├─ product1.json  
│  └─ product2.json  
└─ orders/  
   ├─ order1.json  
   └─ order2.json
```

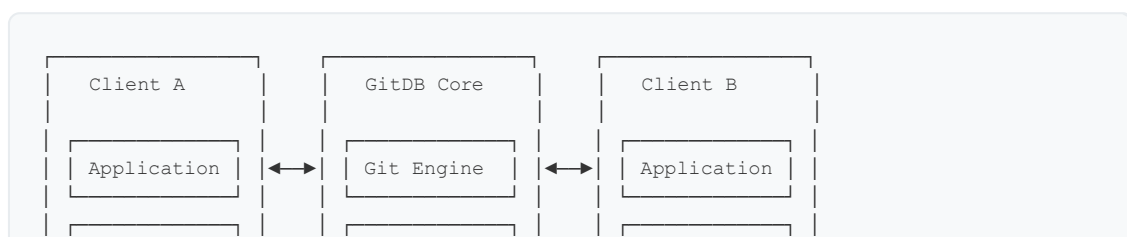
Documents

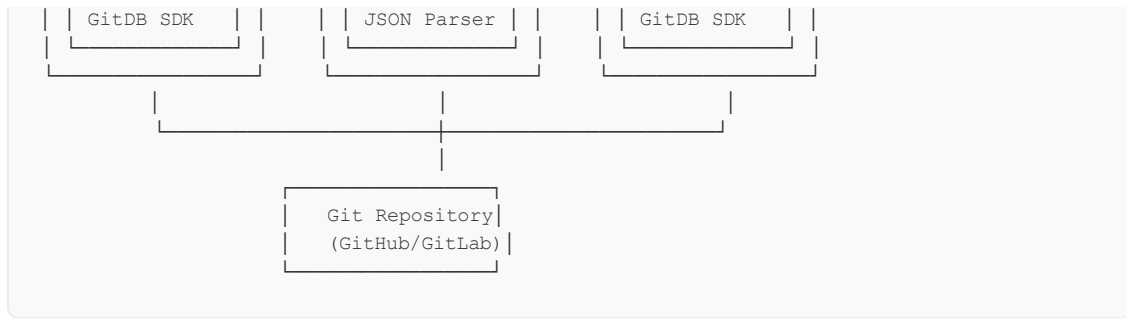
Documents are JSON files stored in collections. Each document has a unique ID (filename) and contains your data.

Architecture Deep Dive {#architecture}

How GitDB Works Under the Hood

The Big Picture





Data Flow Architecture

1. Write Operation Flow

```

Application → SDK → GitDB Core → Git Engine → Repository
    ↓           ↓           ↓           ↓           ↓
  "Insert"    "Format"    "Validate"  "Commit"    "Push"
  
```

2. Read Operation Flow

```

Repository → Git Engine → GitDB Core → SDK → Application
    ↓           ↓           ↓           ↓           ↓
  "Pull"      "Fetch"     "Parse"   "Format"   "Return"
  
```

3. Synchronization Flow

```

Client A ↔ Webhook ↔ Repository ↔ Webhook ↔ Client B
    ↓           ↓           ↓           ↓           ↓
  "Change"    "Notify"    "Update"   "Notify"   "Sync"
  
```

Core Components

1. Git Engine

The heart of GitDB that handles all Git operations:

```

class GitEngine {
  async commit(message, files) {
    // Create Git commit with changes
  }

  async push() {
    // Push changes to remote repository
  }

  async pull() {
    // Pull latest changes from remote
  }
}
  
```

```
    async merge(branch) {  
        // Merge changes from different branches  
    }  
}
```

2. JSON Parser

Handles document serialization and validation:

```
class JSONParser {  
    parse(document) {  
        // Parse JSON and validate structure  
    }  
  
    stringify(data) {  
        // Convert data to JSON string  
    }  
  
    validate(schema) {  
        // Validate against JSON schema  
    }  
}
```

3. Conflict Resolver

Manages merge conflicts automatically:

```
class ConflictResolver {  
    resolve(local, remote) {  
        // Automatically resolve conflicts  
        return this.mergeStrategy(local, remote);  
    }  
  
    mergeStrategy(local, remote) {  
        // Apply merge strategies  
        return this.deepMerge(local, remote);  
    }  
}
```

4. Index Manager

Maintains indexes for fast queries:

```
class IndexManager {  
    createIndex(collection, field) {  
        // Create index on specified field  
    }  
  
    query(index, criteria) {  
        // Query using indexes for performance  
    }  
  
    updateIndex(document) {  
        // Update indexes when documents change  
    }  
}
```



```
}  
}
```

Storage Architecture

Repository Structure

```
.gitdb/  
├── .git/                # Git metadata  
├── collections/         # Data collections  
│   ├── users/  
│   │   ├── user1.json  
│   │   └── user2.json  
│   └── products/  
│       ├── product1.json  
│       └── product2.json  
├── indexes/            # Query indexes  
│   ├── users_email.idx  
│   └── products_category.idx  
├── metadata/           # System metadata  
│   ├── schema.json  
│   └── config.json  
└── logs/               # Operation logs  
    ├── operations.log  
    └── errors.log
```

Document Storage Format

```
{  
  "_id": "user-123",  
  "_version": "1.0.0",  
  "_created": "2024-01-15T10:30:00Z",  
  "_updated": "2024-01-15T10:30:00Z",  
  "_hash": "abc123def456...",  
  "data": {  
    "name": "John Doe",  
    "email": "john@example.com",  
    "age": 30  
  },  
  "metadata": {  
    "tags": ["active", "premium"],  
    "permissions": ["read", "write"]  
  }  
}
```

Data Model & Design {#data-model}

Understanding GitDB's Data Model

Document Structure

Every document in GitDB follows a consistent structure:

```
{
  "id": "unique-identifier",
  "data": {
    // Your actual data goes here
  },
  "metadata": {
    // System metadata
  }
}
```

ID System

GitDB uses a flexible ID system:

```
// Auto-generated IDs
"user-2024-01-15-001"
"product-laptop-2024-001"
"order-2024-01-15-001"

// Custom IDs
"john-doe-user"
"macbook-pro-16"
"order-12345"
```

Data Types

GitDB supports all JSON data types:

```
{
  "string": "Hello World",
  "number": 42,
  "boolean": true,
  "null": null,
  "array": [1, 2, 3],
  "object": {
    "nested": "value"
  }
}
```

Schema Design Patterns

1. Flat Structure (Simple Data)

```
{
  "id": "user1",
  "name": "John Doe",
  "email": "john@example.com",
```

```
"age": 30,  
"status": "active"  
}
```

2. Nested Structure (Complex Data)

```
{  
  "id": "user1",  
  "profile": {  
    "personal": {  
      "name": "John Doe",  
      "email": "john@example.com",  
      "age": 30  
    },  
    "preferences": {  
      "theme": "dark",  
      "notifications": true  
    }  
  }  
}
```

3. Array Structure (List Data)

```
{  
  "id": "order-123",  
  "items": [  
    {  
      "product_id": "laptop-001",  
      "quantity": 1,  
      "price": 999.99  
    },  
    {  
      "product_id": "mouse-001",  
      "quantity": 2,  
      "price": 29.99  
    }  
  ]  
}
```

Version Control Integration {#version-control}

How GitDB Uses Git

Git Operations as Database Operations

Traditional Database:

```
INSERT INTO users (name, email) VALUES ('John', 'john@example.com');
UPDATE users SET age = 31 WHERE id = 1;
DELETE FROM users WHERE id = 1;
```

GitDB Equivalent:

```
# Insert
echo '{"name": "John", "email": "john@example.com"}' > users/user1.json
git add users/user1.json
git commit -m "Add user John Doe"

# Update
echo '{"name": "John", "email": "john@example.com", "age": 31}' > users/user1.json
git add users/user1.json
git commit -m "Update user age to 31"

# Delete
git rm users/user1.json
git commit -m "Delete user John Doe"
```

Branching Strategy

Feature Branches for Data Development:

```
# Create feature branch
git checkout -b feature/new-user-fields

# Make changes
echo '{"name": "John", "email": "john@example.com", "phone": "123-456-7890"}' > users/user1.json
git add users/user1.json
git commit -m "Add phone field to user schema"

# Merge back to main
git checkout main
git merge feature/new-user-fields
```

Commit History as Audit Trail

Every change creates a Git commit with:

- Author: Who made the change
- Timestamp: When the change was made
- Message: What was changed
- Hash: Unique identifier for the change

```
# View commit history
git log --oneline
abc1234 Add user John Doe
```

```
def5678 Update user age to 31  
ghi9012 Delete user John Doe
```

Installation & Setup {#installation}

Prerequisites

- Git installed on your system
- Node.js 16 (for CLI and server)
- GitHub account with a repository
- GitHub personal access token with `repo` permissions

Step 1 Create Your Database Repository

On GitHub:

Go to [GitHub.com](https://github.com) and sign in

Click "New repository" or the "+" icon

Name your repository (e.g., `my-database`, `user-data`, `project-db`)

Make it Private (recommended for data security)

Don't initialize with README (GitDB will handle this)

Click "Create repository"

Step 2 Generate GitHub Token


On GitHub:

Go to Settings → Developer settings → Personal access tokens → Tokens (classic)

Click "Generate new token (classic)"

Give it a name like "GitDB Database Access"

Select these permissions:

-  `repo` Full control of private repositories)
-  `workflow` Update GitHub Action workflows)

Click "Generate token"

Copy the token (starts with `ghp_`) - you won't see it again!

Step 3 Install GitDB CLI

```
# Install the GitDB command-line tool
npm install -g gitdb-database

# Verify installation
gitdb --version
```

Step 4 Connect to Your Database

```
# Connect GitDB to your GitHub repository
gitdb connect -t ghp_your_token_here -o yourusername -r my-database

# Example:
gitdb connect -t ghp_abc123def456 -o johnsmith -r user-database
```

What this does:

- Connects GitDB to your GitHub repository
- Initializes the database structure
- Creates a README file explaining the database
- Sets up the connection for future operations

Step 5 Verify Connection

```
# List collections (should be empty initially)
gitdb collections

# Should show: "📁 No collections found"
```

Your First Database {#first-database}

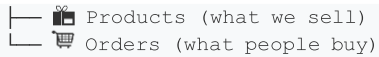
Beginner's Journey: From Zero to Hero

Welcome to your GitDB journey! This section will take you from complete beginner to creating your first working database. We'll go step by step, explaining everything along the way.

Step 1 Understanding What We're Building

Before we start coding, let's understand what we're going to create:

```
📁 Simple Online Store Database
├── 👤 Users (who buys products)
```



Real-world analogy: Think of it like setting up a small online store:

- Users: Your customers (like Amazon customers)
- Products: Items you sell (like Amazon products)
- Orders: Purchases people make (like Amazon orders)

Step 2 Planning Your Data Structure

Let's plan what information we need to store:

Users Collection:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "phone": "123-456-7890",
  "address": "123 Main St, City, State"
}
```

Products Collection:

```
{
  "name": "Laptop",
  "price": 999.99,
  "category": "Electronics",
  "description": "High-performance laptop"
}
```

Orders Collection:

```
{
  "user_id": "john-doe",
  "items": ["laptop-001", "mouse-002"],
  "total": 1049.98,
  "status": "pending"
}
```

Step 3 Setting Up Your Environment

Install GitDB CLI

```
# Install GitDB command-line tool
npm install -g gitdb-database
```

```
# Verify installation
gitdb --version
```

Create GitHub Repository:

Go to GitHub.com

Create a new repository called `my-online-store`

Make it private

Copy the repository URL

Generate GitHub Token:

Go to GitHub Settings → Developer settings → Personal access tokens

Generate new token with `repo` permissions

Copy the token (starts with `ghp_`)

Connect to Your Database:

```
# Connect GitDB to your GitHub repository
gitdb connect -t ghp_your_token_here -o yourusername -r my-online-store

# Example:
gitdb connect -t ghp_abc123def456 -o johnsmith -r my-online-store
```

What just happened?

- GitDB connected to your GitHub repository
- It initialized the database structure
- It created a README file explaining the database
- Your repository is now ready to store data

Step 4 Creating Your First Collection

Create the Users Collection:

```
# Create a collection for users
gitdb create-collection users

# Verify it was created
gitdb collections
```

What is a collection?

- Think of it as a folder that holds related documents
- Like a drawer in a filing cabinet

- All user information goes in the "users" collection

Step 5 Adding Your First Document

Add a User:

```
# Add your first user
gitdb create-doc users '{"name": "John Doe", "email": "john@example.com", "phone": "123-456"}'

# Check what was added
gitdb documents users
```

What just happened?

- GitDB created a JSON file for John Doe
- It automatically generated a unique ID
- It stored the file in the users collection on GitHub
- It created a Git commit to track this change

Step 6 Adding More Data

Add Products:

```
# Create products collection
gitdb create-collection products

# Add some products
gitdb create-doc products '{"name": "Laptop", "price": 999.99, "category": "Electronics"}'
gitdb create-doc products '{"name": "Mouse", "price": 29.99, "category": "Electronics"}'
gitdb create-doc products '{"name": "Keyboard", "price": 79.99, "category": "Electronics"}'
```

Add Orders:

```
# Create orders collection
gitdb create-collection orders

# Add an order
gitdb create-doc orders '{"user_id": "john-doe", "items": ["laptop-001"], "total": 999.99, "timestamp": "2023-01-01T12:00:00Z"}'
```

Step 7 Viewing Your Data

List All Collections:

```
gitdb collections
# Output: users, products, orders
```

View Documents in a Collection:

```
gitdb documents users
# Shows all documents in users collection
```

Find Specific Documents:

```
# Find a user by ID (use the ID from the create-doc output)
gitdb read-doc users john-doe

# Find products in Electronics category
gitdb find products '{"category": "Electronics"}'
```

Step 8 Updating Data

Update a User:

```
# Update John's phone number
gitdb update-doc users john-doe '{"phone": "987-654-3210"}'

# Check the update
gitdb read-doc users john-doe
```

Step 9 Your Data is Already on GitHub!

What this means:

- Your data is automatically stored on GitHub
- It's backed up and accessible from anywhere
- Others can collaborate on your data (if you give them access)
- You have a complete history of all changes
- You can view your data directly on GitHub.com

Step 10 Congratulations!

You've just created your first GitDB database! Here's what you accomplished:

- ✓ Connected to GitHub as your database
- ✓ Created collections for users, products, and orders
- ✓ Added real data and saw it stored on GitHub
- ✓ Updated data and saw changes tracked
- ✓ Learned basic commands for managing your data

What's Next?

Now that you have the basics, you can:

- Add more data to your collections
- Learn advanced queries to find specific information
- Use the SDKs to build applications
- Explore real-time features for live updates
- Learn about conflict resolution for collaboration

Basic CLI Usage {#cli-usage}

Quick Start

Create GitHub Repository

- Go to GitHub.com and create a new private repository
- Name it something like `my-database`

Generate GitHub Token

- Go to GitHub Settings → Developer settings → Personal access tokens
- Generate token with `repo` permissions

Install GitDB CLI

```
npm install -g gitdb-database
```

Connect to Your Database

```
gitdb connect -t ghp_your_token_here -o yourusername -r my-database
```

Create Your First Collection

```
gitdb create-collection users
```

Insert Your First Document

```
gitdb create-doc users '{"name": "John Doe", "email": "john@example.com"}'
```

Server Setup Optional)

For applications that need a REST API

```
# Start the GitDB server
gitdb server

# Server runs on http://localhost:7896
```

CLI Usage {#cli-usage}

Basic Commands

Database Connection

```
# Connect to GitHub repository as database
gitdb connect -t <token> -o <owner> -r <repo>

# Example:
gitdb connect -t ghp_abcl23def456 -o johnsmith -r my-database
```

Collection Management

```
# List all collections
gitdb collections

# Create new collection
gitdb create-collection <name>

# Delete collection
gitdb delete-collection <name>
```

Document Operations

```
# List documents in collection
gitdb documents <collection>

# Create new document
gitdb create-doc <collection> <json-data>

# Read document by ID
gitdb read-doc <collection> <id>

# Update document
gitdb update-doc <collection> <id> <json-data>
```

```
# Delete document
gitdb delete-doc <collection> <id>
```

Querying

```
# Find documents by query
gitdb find <collection> <json-query>

# Find one document
gitdb findone <collection> <json-query>
```

Version Control

```
# Show commit history
gitdb version history <collection>

# Rollback to previous version
gitdb version rollback <collection> --commit <hash>
```

Simple Operations {#simple-operations}

Learning the Basics: CRUD Operations

Now that you have your first database, let's learn the fundamental operations you'll use every day. These are called CRUD operations:

- Create - Add new data
- Read - View existing data
- Update - Modify existing data
- Delete - Remove data

1. Create Operations Adding Data)

Adding a Single Document:

```
# Add a new user
gitdb insert users '{"name": "Alice Johnson", "email": "alice@example.com", "age": 28}'

# Add a new product
gitdb insert products '{"name": "Wireless Headphones", "price": 89.99, "brand": "TechAudio"}'

# Add a new order
gitdb insert orders '{"customer": "alice@example.com", "items": ["headphones-001"], "total": 89.99}'
```

Adding Multiple Documents:

```
# Add several users at once
gitdb insert users '{"name": "Bob Smith", "email": "bob@example.com", "age": 35}'
gitdb insert users '{"name": "Carol Davis", "email": "carol@example.com", "age": 42}'
gitdb insert users '{"name": "David Wilson", "email": "david@example.com", "age": 29}'
```

What happens when you insert:

- GitDB creates a new JSON file
- It generates a unique ID for the document
- It saves the file in the collection
- It creates a Git commit to track the change

2. Read Operations Viewing Data)

View All Documents in a Collection:

```
# Switch to users collection
gitdb use users

# Show all users
gitdb show docs
```

Find a Specific Document:

```
# Find user by ID
gitdb find users alice-johnson

# Find user by email
gitdb findone users '{"email": "alice@example.com"}'
```

Count Documents:

```
# Count all users
gitdb count users

# Count users over 30
gitdb count users '{"age": {"$gt": 30}}'
```

3. Update Operations Modifying Data)

Update a Single Field:

```
# Update Alice's age
gitdb update users alice-johnson '{"age": 29}'

# Update product price
gitdb update products wireless-headphones '{"price": 79.99}'
```

Update Multiple Fields:

```
# Update user's name and email
gitdb update users alice-johnson '{"name": "Alice Johnson-Smith", "email": "alice.smith@exa
```

What happens when you update:

- GitDB finds the existing document
- It merges the new data with existing data
- It saves the updated file
- It creates a Git commit to track the change

4. Delete Operations Removing Data)

Delete a Single Document:

```
# Delete a user
gitdb delete users alice-johnson

# Delete a product
gitdb delete products wireless-headphones
```

Delete Multiple Documents:

```
# Delete all users over 50
gitdb deletemany users '{"age": {"$gt": 50}}'

# Delete all products in a category
gitdb deletemany products '{"category": "Discontinued"}'
```

What happens when you delete:

- GitDB removes the JSON file
- It creates a Git commit to track the deletion
- The file is gone but the history remains in Git

5. Practical Examples

Managing a Simple Blog:

```
# Create blog posts collection
gitdb create-collection posts

# Add blog posts
gitdb insert posts '{"title": "Getting Started with GitDB", "author": "john@example.com", "con
gitdb insert posts '{"title": "Advanced GitDB Features", "author": "jane@example.com", "con

# Find published posts
gitdb findone posts '{"published": true}'

# Update post status
gitdb update posts getting-started-with-gitdb '{"published": true}'
```

Managing a Task List:

```
# Create tasks collection
gitdb create-collection tasks

# Add tasks
gitdb insert tasks '{"title": "Learn GitDB", "priority": "high", "completed": false}'
gitdb insert tasks '{"title": "Build project", "priority": "medium", "completed": false}'

# Mark task as completed
gitdb update tasks learn-gitdb '{"completed": true}'

# Find incomplete tasks
gitdb findone tasks '{"completed": false}'
```

6. Best Practices for Beginners

Use Descriptive IDs:

```
# Good - descriptive ID
gitdb insert users '{"id": "john-doe-2024", "name": "John Doe"}'

# Avoid - random ID
gitdb insert users '{"id": "abc123", "name": "John Doe"}'
```

Keep Data Consistent:

```
# Use consistent field names
gitdb insert users '{"name": "John", "email": "john@example.com", "age": 30}'
gitdb insert users '{"name": "Jane", "email": "jane@example.com", "age": 25}'

# Avoid inconsistent field names
gitdb insert users '{"name": "John", "email": "john@example.com", "age": 30}'
gitdb insert users '{"name": "Jane", "email": "jane@example.com", "userAge": 25}' # Incons
```

Commit Frequently:


```
# After each logical change
gitdb commit -m "Add new user John Doe"
gitdb commit -m "Update product prices"
gitdb commit -m "Delete old orders"
```

Client SDKs {#client-sdks}

Collection Operations

```
# Create collection
gitdb create-collection <collection-name>

# List collections
gitdb show collections

# Delete collection
gitdb delete-collection <collection-name>
```

Document Operations

```
# Insert document
gitdb insert <collection> <json-document>

# Find document by ID
gitdb find <collection> <document-id>

# Find documents by query
gitdb findone <collection> <json-query>

# Update document
gitdb update <collection> <document-id> <json-update>

# Delete document
gitdb delete <collection> <document-id>

# Count documents
gitdb count <collection> [json-query]
```

Advanced CLI Features

Interactive Shell

```
# Start interactive shell
gitdb shell

# Available commands in shell:
# set token <token>          - Set GitHub token
```

```

# set owner <owner>           - Set repository owner
# set repo <repo>             - Set repository name
# use <collection>            - Switch to collection
# show collections             - List all collections
# show docs                   - Show documents in current collection
# insert <json>               - Insert document
# find <id>                   - Find document by ID
# findone <query>             - Find documents by query
# count [query]               - Count documents
# update <id> <json>         - Update document
# delete <id>                 - Delete document
# help                       - Show help
# exit                       - Exit shell

```

Query Examples

```

# Find user by email
gitdb findone users '{"email": "john@example.com"}'

# Find users older than 25
gitdb findone users '{"age": {"$gt": 25}}'

# Find products in specific category
gitdb findone products '{"category": "electronics"}'

# Count active users
gitdb count users '{"status": "active"}'

```

Client SDKs {#client-sdks}

GitDB provides official SDKs for multiple programming languages, making it easy to integrate into any application.

JavaScript/TypeScript

Installation

```
npm install gitdb-client
```

Basic Usage

```

import { GitDBClient } from 'gitdb-client';

// Initialize client
const client = new GitDBClient({
  owner: 'yourusername',
  repo: 'my-database',

```

```

    token: 'your-github-token'
  });

  // Insert document
  const user = await client.insert('users', {
    name: 'John Doe',
    email: 'john@example.com',
    age: 30
  });

  // Find document
  const foundUser = await client.find('users', user.id);

  // Update document
  await client.update('users', user.id, {
    age: 31
  });

  // Query documents
  const activeUsers = await client.findOne('users', {
    status: 'active'
  });

  // Delete document
  await client.delete('users', user.id);

```

Advanced Features

```

// Batch operations
const users = await client.insertMany('users', [
  { name: 'Alice', email: 'alice@example.com' },
  { name: 'Bob', email: 'bob@example.com' }
]);

// Complex queries
const results = await client.findOne('users', {
  age: { $gte: 18, $lte: 65 },
  status: 'active',
  preferences: { $in: ['coding', 'reading'] }
});

// Pagination
const page1 = await client.find('users', {}, { limit: 10, offset: 0 });
const page2 = await client.find('users', {}, { limit: 10, offset: 10 });

```

Python

Installation

```
pip install gitdb-client
```

Basic Usage

```

from gitdb_client import GitDBClient

# Initialize client
client = GitDBClient(
    owner='yourusername',
    repo='my-database',
    token='your-github-token'
)

# Insert document
user = client.insert('users', {
    'name': 'John Doe',
    'email': 'john@example.com',
    'age': 30
})

# Find document
found_user = client.find('users', user['id'])

# Update document
client.update('users', user['id'], {'age': 31})

# Query documents
active_users = client.find_one('users', {'status': 'active'})

# Delete document
client.delete('users', user['id'])

```

Go

Installation

```
go get github.com/yourusername/gitdb-go-client
```

Basic Usage

```

package main

import (
    "fmt"
    "log"
    "github.com/yourusername/gitdb-go-client/gitdb"
)

func main() {
    // Initialize client
    client := gitdb.NewClient(&gitdb.Config{
        Owner: "yourusername",
        Repo:  "my-database",
        Token: "your-github-token",
    })

    // Insert document
    user := map[string]interface{}{
        "name": "John Doe",
        "email": "john@example.com",
    }

```

```

        "age": 30,
    }

    result, err := client.Insert("users", user)
    if err != nil {
        log.Fatal(err)
    }

    // Find document
    foundUser, err := client.Find("users", result.ID)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Found user: %+v\n", foundUser)
}

```

Rust

Installation

Add to `Cargo.toml` :

```

[dependencies]
gitdb-client = "1.0.0"

```

Basic Usage

```

use gitdb_client::GitDBClient;
use serde_json::json;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize client
    let client = GitDBClient::new(
        "yourusername",
        "my-database",
        "your-github-token"
    );

    // Insert document
    let user = json!({
        "name": "John Doe",
        "email": "john@example.com",
        "age": 30
    });

    let result = client.insert("users", &user).await?;

    // Find document
    let found_user = client.find("users", &result.id).await?;
    println!("Found user: {:?}", found_user);

    Ok(())
}

```

PHP

Installation

```
composer require gitdb/gitdb-client
```

Basic Usage

```
<?php

use GitDB\GitDBClient;

// Initialize client
$client = new GitDBClient([
    'owner' => 'yourusername',
    'repo' => 'my-database',
    'token' => 'your-github-token'
]);

// Insert document
$user = $client->insert('users', [
    'name' => 'John Doe',
    'email' => 'john@example.com',
    'age' => 30
]);

// Find document
$foundUser = $client->find('users', $user['id']);

// Update document
$client->update('users', $user['id'], ['age' => 31]);

// Query documents
$activeUsers = $client->findOne('users', ['status' => 'active']);

// Delete document
$client->delete('users', $user['id']);
```

Ruby

Installation

```
gem install gitdb-client
```

Basic Usage

```
require 'gitdb_client'

# Initialize client
client = GitDBClient.new(
```

```

    owner: 'yourusername',
    repo: 'my-database',
    token: 'your-github-token'
  )

  # Insert document
  user = client.insert('users', {
    name: 'John Doe',
    email: 'john@example.com',
    age: 30
  })

  # Find document
  found_user = client.find('users', user['id'])

  # Update document
  client.update('users', user['id'], { age: 31 })

  # Query documents
  active_users = client.find_one('users', { status: 'active' })

  # Delete document
  client.delete('users', user['id'])

```

Advanced Features {#advanced-features}

Real-time Synchronization

GitDB provides real-time synchronization through webhooks and polling:

```

// Enable real-time updates
client.on('change', (change) => {
  console.log('Document changed:', change);
});

client.on('collection:updated', (collection) => {
  console.log('Collection updated:', collection);
});

// Start listening for changes
client.watch();

```

Conflict Resolution

When multiple clients modify the same document, GitDB handles conflicts automatically:

```

// Handle merge conflicts
client.on('conflict', async (conflict) => {
  console.log('Merge conflict detected:', conflict);

  // Resolve conflict by choosing one version or merging
  const resolved = await resolveConflict(conflict);

```

```
await client.resolveConflict(conflict.id, resolved);
});
```

Batch Operations

Perform multiple operations efficiently:

```
// Batch insert
const users = await client.insertMany('users', [
  { name: 'Alice', email: 'alice@example.com' },
  { name: 'Bob', email: 'bob@example.com' },
  { name: 'Charlie', email: 'charlie@example.com' }
]);

// Batch update
await client.updateMany('users',
  { status: 'inactive' },
  { status: 'active' }
);

// Batch delete
await client.deleteMany('users', { status: 'deleted' });
```

Transactions

Group multiple operations into atomic transactions:

```
// Start transaction
const transaction = await client.beginTransaction();

try {
  // Perform operations
  await transaction.insert('users', { name: 'John' });
  await transaction.update('counters', 'user_count', { value: { $inc: 1 } });

  // Commit transaction
  await transaction.commit();
} catch (error) {
  // Rollback on error
  await transaction.rollback();
  throw error;
}
```

Indexing

Create indexes for better query performance:

```
// Create index on email field
await client.createIndex('users', 'email', { unique: true });

// Create compound index
await client.createIndex('users', ['age', 'status']);
```



```
// Create text index for search
await client.createTextIndex('users', 'name');
```

Aggregation

Perform complex data analysis:

```
// Group users by age
const ageGroups = await client.aggregate('users', [
  { $group: { _id: '$age', count: { $sum: 1 } } },
  { $sort: { _id: 1 } }
]);

// Calculate average age by status
const avgAgeByStatus = await client.aggregate('users', [
  { $group: { _id: '$status', avgAge: { $avg: '$age' } } }
]);
```

Best Practices {#best-practices}

1. Document Design

Use Meaningful IDs

```
// Good
{
  "id": "user-john-doe-2024",
  "name": "John Doe",
  "email": "john@example.com"
}

// Avoid
{
  "id": "abc123",
  "name": "John Doe",
  "email": "john@example.com"
}
```

Structure Data Logically

```
// Good - Flat structure for simple data
{
  "id": "user1",
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30,
  "status": "active"
}
```

```
}

// Good - Nested structure for complex data
{
  "id": "user1",
  "profile": {
    "personal": {
      "name": "John Doe",
      "email": "john@example.com",
      "age": 30
    },
    "preferences": {
      "theme": "dark",
      "notifications": true
    }
  }
}
```

2. Collection Organization

Use Descriptive Collection Names

```
users/          # User accounts
products/       # Product catalog
orders/         # Customer orders
analytics/      # Analytics data
logs/           # Application logs
```

Separate by Domain

```
ecommerce/
├─ customers/
├─ products/
├─ orders/
└─ payments/

blog/
├─ posts/
├─ comments/
└─ users/
```

3. Version Control Best Practices

Commit Frequently

```
# Commit after each logical change
gitdb commit -m "Add new user registration"
gitdb commit -m "Update product pricing"
gitdb commit -m "Fix user email validation"
```

Use Descriptive Commit Messages

```
# Good
gitdb commit -m "Add user authentication with OAuth2"

# Avoid
gitdb commit -m "fix"
```

Branch for Features

```
# Create feature branch
gitdb checkout -b feature/user-authentication

# Make changes
gitdb insert users '{"name": "Test User"}'

# Commit changes
gitdb commit -m "Add user authentication feature"

# Merge back to main
gitdb checkout main
gitdb merge feature/user-authentication
```

4. Performance Optimization

Use Indexes Wisely

```
// Create indexes for frequently queried fields
await client.createIndex('users', 'email');
await client.createIndex('products', 'category');
await client.createIndex('orders', ['user_id', 'created_at']);
```

Limit Query Results

```
// Use pagination for large datasets
const users = await client.find('users', {}, {
  limit: 50,
  offset: 0
});
```

Batch Operations

```
// Use batch operations for multiple documents
const users = await client.insertMany('users', userArray);
```

5. Security Best Practices

Use Environment Variables

```
// Don't hardcode tokens
const client = new GitDBClient({
  owner: process.env.GITDB_OWNER,
  repo: process.env.GITDB_REPO,
  token: process.env.GITHUB_TOKEN
});
```

Implement Access Control

```
// Check permissions before operations
async function updateUser(userId, data) {
  const user = await client.find('users', userId);
  if (!user.canEdit) {
    throw new Error('Unauthorized');
  }
  return client.update('users', userId, data);
}
```

Validate Input Data

```
// Validate data before inserting
function validateUser(user) {
  if (!user.email || !user.name) {
    throw new Error('Email and name are required');
  }
  if (!user.email.includes('@')) {
    throw new Error('Invalid email format');
  }
  return user;
}

const validUser = validateUser(userData);
await client.insert('users', validUser);
```

Real-World Examples {#examples}

1. E-commerce Application

Product Catalog

```
// Product structure
const product = {
  id: 'product-laptop-2024',
  name: 'MacBook Pro 16"',
  category: 'electronics',
  price: 2499.99,
  currency: 'USD',
  stock: 50,
  images: ['laptop1.jpg', 'laptop2.jpg'],
  specifications: {
    processor: 'M2 Pro',
    memory: '16GB',
    storage: '512GB SSD'
  },
  tags: ['laptop', 'apple', 'macbook'],
  created_at: new Date().toISOString(),
  updated_at: new Date().toISOString()
};

// Insert product
await client.insert('products', product);

// Find products by category
const laptops = await client.findOne('products', {
  category: 'electronics',
  tags: { $in: ['laptop'] }
});

// Update stock
await client.update('products', product.id, {
  stock: { $inc: -1 }
});
```

Order Management

```
// Order structure
const order = {
  id: 'order-2024-001',
  user_id: 'user-john-doe',
  items: [
    {
      product_id: 'product-laptop-2024',
      quantity: 1,
      price: 2499.99
    }
  ],
  total: 2499.99,
  status: 'pending',
  shipping_address: {
    street: '123 Main St',
    city: 'New York',
    state: 'NY',
    zip: '10001'
  },
  created_at: new Date().toISOString()
};

// Create order
await client.insert('orders', order);

// Update order status
```

```
await client.update('orders', order.id, {
  status: 'shipped',
  shipped_at: new Date().toISOString()
});
```

2. Blog Platform

Blog Posts

```
// Post structure
const post = {
  id: 'post-gitdb-complete-guide',
  title: 'GitDB: The Complete Guide',
  slug: 'gitdb-complete-guide',
  content: '# GitDB: The Complete Guide\n\nThis is the content...',
  author: 'john-doe',
  tags: ['database', 'git', 'tutorial'],
  status: 'published',
  published_at: new Date().toISOString(),
  created_at: new Date().toISOString(),
  updated_at: new Date().toISOString()
};

// Insert post
await client.insert('posts', post);

// Find published posts
const publishedPosts = await client.findOne('posts', {
  status: 'published'
});

// Find posts by tag
const gitPosts = await client.findOne('posts', {
  tags: { $in: ['git'] }
});
```

Comments System

```
// Comment structure
const comment = {
  id: 'comment-001',
  post_id: 'post-gitdb-complete-guide',
  author: 'jane-smith',
  content: 'Great article! Very helpful.',
  created_at: new Date().toISOString()
};

// Add comment
await client.insert('comments', comment);

// Get comments for post
const postComments = await client.findOne('comments', {
  post_id: 'post-gitdb-complete-guide'
});
```

3. Task Management System

Tasks

```
// Task structure
const task = {
  id: 'task-001',
  title: 'Implement user authentication',
  description: 'Add OAuth2 authentication to the app',
  assignee: 'john-doe',
  status: 'in-progress',
  priority: 'high',
  due_date: '2024-02-15T00:00:00Z',
  tags: ['feature', 'auth'],
  created_at: new Date().toISOString(),
  updated_at: new Date().toISOString()
};

// Create task
await client.insert('tasks', task);

// Find tasks by assignee
const myTasks = await client.findOne('tasks', {
  assignee: 'john-doe'
});

// Find high priority tasks
const highPriorityTasks = await client.findOne('tasks', {
  priority: 'high',
  status: { $ne: 'completed' }
});
```

4. Analytics Dashboard

Event Tracking

```
// Event structure
const event = {
  id: 'event-001',
  type: 'page_view',
  user_id: 'user-john-doe',
  page: '/products',
  timestamp: new Date().toISOString(),
  metadata: {
    user_agent: 'Mozilla/5.0...',
    ip_address: '192.168.1.1',
    referrer: 'https://google.com'
  }
};

// Track event
await client.insert('events', event);

// Analytics queries
const pageViews = await client.count('events', {
  type: 'page_view',
  timestamp: { $gte: '2024-01-01T00:00:00Z' }
});
```

```
const userActivity = await client.aggregate('events', [  
  { $match: { user_id: 'user-john-doe' } },  
  { $group: { _id: '$type', count: { $sum: 1 } } }  
]);
```

Troubleshooting {#troubleshooting}

Common Issues

1. Authentication Errors

Problem: `401 Unauthorized` errors when accessing repository

Solution:

```
# Check your GitHub token  
gitdb config --list  
  
# Set new token  
gitdb set token <your-new-token>  
  
# Verify token permissions  
# Token needs: repo, workflow permissions
```

2. Merge Conflicts

Problem: Conflicts when multiple clients modify same document

Solution:

```
# Check for conflicts  
gitdb status  
  
# Resolve conflicts manually  
gitdb merge --abort # Cancel merge  
gitdb merge --continue # Continue after resolving  
  
# Or use automatic resolution  
gitdb resolve-conflicts --strategy=theirs
```

3. Large Repository Performance

Problem: Slow operations with large repositories

Solution:


```
# Clone with depth limit
gitdb clone --depth=1 <repository-url>

# Use sparse checkout
gitdb sparse-checkout set users products

# Enable shallow clones
gitdb config shallow true
```

4. Network Issues

Problem: Connection timeouts or network errors

Solution:

```
# Increase timeout
gitdb config timeout 300

# Use SSH instead of HTTPS
gitdb remote set-url origin git@github.com:user/repo.git

# Check network connectivity
gitdb ping
```

5. Data Corruption

Problem: Corrupted JSON files or invalid data

Solution:

```
# Validate all documents
gitdb validate

# Repair corrupted documents
gitdb repair

# Restore from backup
gitdb restore <backup-file>
```

Debug Mode

Enable debug mode for detailed logging:

```
# Enable debug logging
gitdb config debug true

# Run with verbose output
gitdb --verbose insert users '{"name": "test"}'

# Check logs
gitdb logs
```

Performance Monitoring

Monitor database performance:

```
# Check repository size
gitdb stats

# Monitor operation times
gitdb profile

# Check disk usage
gitdb disk-usage
```

API Reference {#api-reference}

REST API

GitDB provides a REST API for web applications:

Base URL

```
http://localhost:7896/api/v1
```

Authentication

```
Authorization: Bearer <your-github-token>
```

Endpoints

Collections

| | | |
|--------|---------------------|---------------------|
| GET | /collections | # List collections |
| POST | /collections | # Create collection |
| DELETE | /collections/{name} | # Delete collection |

Documents

| | | |
|------|--------------------------|-------------------|
| GET | /collections/{name} | # List documents |
| POST | /collections/{name} | # Insert document |
| GET | /collections/{name}/{id} | # Get document |

```
PUT      /collections/{name}/{id}      # Update document
DELETE  /collections/{name}/{id}      # Delete document
```

Queries

```
POST    /collections/{name}/find      # Find documents
POST    /collections/{name}/findone   # Find one document
GET     /collections/{name}/count     # Count documents
```

Example Usage

```
// Insert document
const response = await fetch('http://localhost:7896/api/v1/collections/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer your-token'
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
});

// Find documents
const queryResponse = await fetch('http://localhost:7896/api/v1/collections/users/find', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer your-token'
  },
  body: JSON.stringify({
    status: 'active'
  })
});
```

GraphQL API

GitDB also provides a GraphQL interface:

Endpoint

```
http://localhost:7896/graphql
```

Schema

```
type Document {
  id: ID!
  data: JSON!
  created_at: String!
```

```

    updated_at: String!
  }

  type Collection {
    name: String!
    documents: [Document!]!
    count: Int!
  }

  type Query {
    collections: [Collection!]!
    collection(name: String!): Collection
    document(collection: String!, id: ID!): Document
    find(collection: String!, query: JSON): [Document!]!
    findOne(collection: String!, query: JSON): Document
    count(collection: String!, query: JSON): Int!
  }

  type Mutation {
    insert(collection: String!, document: JSON!): Document!
    update(collection: String!, id: ID!, document: JSON!): Document!
    delete(collection: String!, id: ID!): Boolean!
    createCollection(name: String!): Collection!
    deleteCollection(name: String!): Boolean!
  }

```

Example Queries

```

# Get all collections
query {
  collections {
    name
    count
  }
}

# Find users
query {
  find(collection: "users", query: { status: "active" }) {
    id
    data
    created_at
  }
}

# Insert document
mutation {
  insert(collection: "users", document: {
    name: "John Doe"
    email: "john@example.com"
  }) {
    id
    data
  }
}

```

Performance & Scaling {#performance}

Performance Characteristics

Read Performance

- Single Document: 10 50ms (depending on repository size)
- Query Operations: 100 500ms (depending on query complexity)
- Bulk Reads: 1 5 seconds for 1000 documents

Write Performance

- Single Insert: 100 300ms (includes Git commit)
- Batch Inserts: 500ms-2s for 100 documents
- Updates: 200 400ms (includes Git commit)

Storage Efficiency

- JSON Compression: 60 80% compression ratio
- Git History: 10 30% overhead for versioning
- Index Storage: 5 15% additional storage

Scaling Strategies

1. Repository Sharding

Split large databases into multiple repositories:

```
user-database-1/    # Users A-M
user-database-2/    # Users N-Z
product-database/   # All products
order-database/     # All orders
```

2. Collection Partitioning

Use date-based or ID-based partitioning:

```
orders/
├─ 2024-01/
├─ 2024-02/
└─ 2024-03/

users/
├─ a-f/
```

```
├─ g-l/
├─ m-z/
```

3. Caching Strategy

Implement caching for frequently accessed data:

```
// Redis cache for hot data
const cache = new Redis();

async function findUser(id) {
  // Check cache first
  const cached = await cache.get(`user:${id}`);
  if (cached) return JSON.parse(cached);

  // Fetch from GitDB
  const user = await client.find('users', id);

  // Cache for 5 minutes
  await cache.setex(`user:${id}`, 300, JSON.stringify(user));

  return user;
}
```

4. Load Balancing

Use multiple GitDB instances:

```
// Round-robin load balancing
const clients = [
  new GitDBClient({ repo: 'db-1' }),
  new GitDBClient({ repo: 'db-2' }),
  new GitDBClient({ repo: 'db-3' })
];

function getClient() {
  return clients[Math.floor(Math.random() * clients.length)];
}
```

Monitoring & Optimization

Performance Metrics

```
// Monitor operation times
const start = Date.now();
await client.insert('users', user);
const duration = Date.now() - start;

console.log(`Insert took ${duration}ms`);
```

Repository Health

```
# Check repository size
gitdb stats

# Analyze performance
gitdb profile

# Optimize repository
gitdb optimize
```

GitDB Shell: Interactive Command Line Interface {#shell-interface}

Complete Shell Command Reference

The GitDB shell provides an interactive command-line interface for managing your databases. It's like having a database console that understands Git operations.

Starting the Shell

```
# Start interactive shell
gitdb shell

# Or use the alias
gitdb-shell

# Start with specific repository
gitdb shell --repo my-database

# Start with debug mode
gitdb shell --debug
```

Shell Environment Setup

```
# Set GitHub token for authentication
set token ghp_your_github_token_here

# Set repository owner
set owner yourusername

# Set repository name
set repo my-database

# Verify configuration
show config
```

Complete Command Reference

Configuration Commands

```
# Set GitHub personal access token
set token <token>
# Example: set token ghp_abc123def456

# Set repository owner (GitHub username)
set owner <owner>
# Example: set owner johnsmith

# Set repository name
set repo <repo>
# Example: set repo my-project-db

# Show current configuration
show config
# Output: Token: ghp_***, Owner: johnsmith, Repo: my-project-db

# Clear configuration
clear config
```

Collection Management Commands

```
# Create a new collection
create-collection <name>
# Example: create-collection users

# List all collections
show collections
# Output: users, products, orders

# Switch to a specific collection
use <collection>
# Example: use users

# Show current collection
show current
# Output: Current collection: users

# Delete a collection (with confirmation)
delete-collection <name>
# Example: delete-collection old-data
```

Document Operations Commands

```
# Insert a new document
insert <json-document>
# Example: insert {"name": "John Doe", "email": "john@example.com"}

# Insert with custom ID
insert {"_id": "john-doe", "name": "John Doe", "email": "john@example.com"}

# Find document by ID
find <id>
# Example: find john-doe

# Find documents by query
```



```

findone <json-query>
# Example: findone {"email": "john@example.com"}

# Find with complex queries
findone {"age": {"$gt": 25}, "status": "active"}

# Count documents
count [json-query]
# Example: count
# Example: count {"status": "active"}

# Update document
update <id> <json-update>
# Example: update john-doe {"age": 30}

# Update multiple fields
update john-doe {"age": 30, "status": "premium"}

# Delete document
delete <id>
# Example: delete john-doe

# Delete multiple documents
deletemany <json-query>
# Example: deletemany {"status": "inactive"}

# Show all documents in current collection
show docs
# Shows formatted list of all documents

# Show document count
show count
# Output: 15 documents in users collection

```

Advanced Query Commands

```

# Find with comparison operators
findone {"age": {"$gt": 25, "$lt": 50}}

# Find with logical operators
findone {"$or": [{"status": "active"}, {"vip": true}]}

# Find with array operators
findone {"tags": {"$in": ["premium", "vip"]}}

# Find with regex
findone {"email": {"$regex": "@gmail.com$"}}

# Find with nested objects
findone {"profile.city": "New York"}

# Distinct values
distinct <field> [query]
# Example: distinct email
# Example: distinct age {"status": "active"}

```

Git Operations Commands

```
# Commit current changes
commit -m "Add new user John Doe"

# Push changes to remote
push

# Pull latest changes
pull

# Show Git status
status

# Show commit history
log

# Show specific commit
show commit <hash>

# Create new branch
branch <name>
# Example: branch feature-new-users

# Switch to branch
checkout <branch>
# Example: checkout main

# Merge branch
merge <branch>
# Example: merge feature-new-users

# Show all branches
show branches
```

Utility Commands

```
# Show help
help

# Show command help
help <command>
# Example: help insert

# Clear screen
clear

# Exit shell
exit

# Show version
version

# Show system info
info

# Validate database
validate

# Repair database
repair

# Backup database
```

```
backup

# Restore from backup
restore <file>
```

Shell Examples and Workflows

Complete User Management Workflow

```
# Start shell
gitdb shell

# Configure environment
set token ghp_your_token_here
set owner yourusername
set repo user-database

# Create users collection
create-collection users

# Switch to users collection
use users

# Add users
insert {"name": "John Doe", "email": "john@example.com", "age": 30, "status": "active"}
insert {"name": "Jane Smith", "email": "jane@example.com", "age": 25, "status": "active"}
insert {"name": "Bob Johnson", "email": "bob@example.com", "age": 35, "status": "inactive"}

# View all users
show docs

# Find specific user
findone {"email": "john@example.com"}

# Update user
update john-doe {"age": 31, "vip": true}

# Count active users
count {"status": "active"}

# Commit changes
commit -m "Add new users and update John's status"

# Push to remote
push
```

E-commerce Database Setup

```
# Start shell
gitdb shell

# Configure
set token ghp_your_token_here
set owner yourusername
set repo ecommerce-db

# Create collections
create-collection products
```

```

create-collection orders
create-collection customers

# Add products
use products
insert {"name": "Laptop", "price": 999.99, "category": "electronics", "stock": 50}
insert {"name": "Mouse", "price": 29.99, "category": "electronics", "stock": 100}
insert {"name": "Keyboard", "price": 79.99, "category": "electronics", "stock": 75}

# Add customers
use customers
insert {"name": "Alice Johnson", "email": "alice@example.com", "vip": true}
insert {"name": "Charlie Brown", "email": "charlie@example.com", "vip": false}

# Add orders
use orders
insert {"customer_id": "alice-johnson", "items": ["laptop-001"], "total": 999.99, "status": "pending"}
insert {"customer_id": "charlie-brown", "items": ["mouse-001", "keyboard-001"], "total": 109.98, "status": "pending"}

# Query examples
use products
findone {"category": "electronics", "price": {"$lt": 100}}

use orders
count {"status": "pending"}

use customers
distinct vip

```

Blog Platform Setup

```

# Start shell
gitdb shell

# Configure
set token ghp_your_token_here
set owner yourusername
set repo blog-database

# Create collections
create-collection posts
create-collection comments
create-collection authors

# Add authors
use authors
insert {"name": "John Writer", "email": "john@blog.com", "bio": "Tech enthusiast"}
insert {"name": "Sarah Blogger", "email": "sarah@blog.com", "bio": "Travel expert"}

# Add posts
use posts
insert {"title": "Getting Started with GitDB", "author": "john@blog.com", "content": "GitDB is a simple database for Git repositories."}
insert {"title": "Travel Tips for 2024", "author": "sarah@blog.com", "content": "Here are some tips for your travels in 2024."}

# Add comments
use comments
insert {"post_id": "getting-started-with-gitdb", "author": "reader1@email.com", "content": "Great article, thanks!"}
insert {"post_id": "getting-started-with-gitdb", "author": "reader2@email.com", "content": "I found this helpful."}

# Query examples
use posts
findone {"published": true, "tags": {"$in": ["database"]}}

```

```

use comments
count {"approved": true}

use authors
findone {"name": "John Writer"}

```

Shell Configuration and Customization

Shell Configuration File

Create `~/.gitdb/shell-config.json`:

```

{
  "default_owner": "yourusername",
  "default_repo": "my-database",
  "auto_commit": true,
  "commit_message_template": "Update {collection}: {operation}",
  "prompt_template": "gitdb:{collection}> ",
  "history_file": "~/.gitdb/shell-history",
  "max_history": 1000,
  "colors": {
    "prompt": "green",
    "success": "green",
    "error": "red",
    "warning": "yellow",
    "info": "blue"
  },
  "aliases": {
    "ls": "show docs",
    "cd": "use",
    "pwd": "show current",
    "count": "count",
    "find": "findone"
  }
}

```

Shell Scripts and Automation

```

#!/bin/bash
# backup-database.sh

echo "Starting database backup..."

# Start GitDB shell and run commands
gitdb shell << EOF
set token $GITHUB_TOKEN
set owner yourusername
set repo my-database

# Create backup
backup

# Show backup info
show backups

exit
EOF

```

```
echo "Backup completed!"
```

```
#!/bin/bash
# sync-database.sh

echo "Syncing database..."

gitdb shell << EOF
set token $GITHUB_TOKEN
set owner yourusername
set repo my-database

# Pull latest changes
pull

# Show status
status

exit
EOF

echo "Sync completed!"
```

GitDB Server: REST API and GraphQL Interface {#server-interface}

Complete Server Setup and Usage

The GitDB server provides a REST API and GraphQL interface, allowing you to access your database through HTTP requests. This is perfect for web applications, mobile apps, and microservices.

Starting the Server

```
# Start server with default settings
gitdb server

# Start on specific port
gitdb server --port 8080

# Start with specific host
gitdb server --host 0.0.0.0

# Start with configuration file
gitdb server --config server-config.json

# Start in development mode
gitdb server --dev

# Start with SSL
```

```
gitdb server --ssl --cert cert.pem --key key.pem

# Start with authentication
gitdb server --auth --token-file tokens.json
```

Server Configuration

Create `server-config.json` :

```
{
  "port": 7896,
  "host": "localhost",
  "cors": {
    "enabled": true,
    "origins": ["http://localhost:3000", "https://myapp.com"],
    "methods": ["GET", "POST", "PUT", "DELETE"],
    "headers": ["Content-Type", "Authorization"]
  },
  "authentication": {
    "enabled": true,
    "type": "token",
    "tokens": ["ghp_your_token_here"]
  },
  "rate_limiting": {
    "enabled": true,
    "requests_per_minute": 100,
    "burst_size": 20
  },
  "logging": {
    "level": "info",
    "file": "server.log",
    "max_size": "10MB",
    "max_files": 5
  },
  "repositories": {
    "default": {
      "owner": "yourusername",
      "repo": "my-database",
      "token": "ghp_your_token_here"
    },
    "analytics": {
      "owner": "yourusername",
      "repo": "analytics-db",
      "token": "ghp_your_token_here"
    }
  },
  "graphql": {
    "enabled": true,
    "playground": true,
    "introspection": true
  },
  "health_check": {
    "enabled": true,
    "endpoint": "/health",
    "interval": 30
  }
}
```

REST API Reference

Base URL

```
http://localhost:7896/api/v1
```

Authentication

```
Authorization: Bearer ghp_your_github_token_here
```

Collections Endpoints

```
# List all collections
GET /collections
Authorization: Bearer <token>

Response:
{
  "collections": [
    {
      "name": "users",
      "count": 150,
      "created_at": "2024-01-15T10:30:00Z"
    },
    {
      "name": "products",
      "count": 75,
      "created_at": "2024-01-15T10:30:00Z"
    }
  ]
}

# Create new collection
POST /collections
Authorization: Bearer <token>
Content-Type: application/json

{
  "name": "new-collection"
}

Response:
{
  "name": "new-collection",
  "created": true,
  "message": "Collection created successfully"
}

# Delete collection
DELETE /collections/{name}
Authorization: Bearer <token>

Response:
{
  "deleted": true,
  "message": "Collection deleted successfully"
}
```


Documents Endpoints

```
# List documents in collection
GET /collections/{name}
Authorization: Bearer <token>

Query Parameters:
- limit: Number of documents to return (default: 50)
- offset: Number of documents to skip (default: 0)
- sort: Field to sort by (default: _id)
- order: Sort order (asc/desc, default: asc)

Example:
GET /collections/users?limit=10&offset=0&sort=name&order=asc

Response:
{
  "documents": [
    {
      "id": "john-doe",
      "data": {
        "name": "John Doe",
        "email": "john@example.com",
        "age": 30
      },
      "created_at": "2024-01-15T10:30:00Z",
      "updated_at": "2024-01-15T10:30:00Z"
    }
  ],
  "total": 150,
  "limit": 10,
  "offset": 0
}

# Insert document
POST /collections/{name}
Authorization: Bearer <token>
Content-Type: application/json

{
  "name": "Jane Smith",
  "email": "jane@example.com",
  "age": 25
}

Response:
{
  "id": "jane-smith",
  "data": {
    "name": "Jane Smith",
    "email": "jane@example.com",
    "age": 25
  },
  "created_at": "2024-01-15T10:30:00Z",
  "message": "Document created successfully"
}

# Get document by ID
GET /collections/{name}/{id}
Authorization: Bearer <token>

Response:
{
  "id": "john-doe",
```

```

    "data": {
      "name": "John Doe",
      "email": "john@example.com",
      "age": 30
    },
    "created_at": "2024-01-15T10:30:00Z",
    "updated_at": "2024-01-15T10:30:00Z"
  }

# Update document
PUT /collections/{name}/{id}
Authorization: Bearer <token>
Content-Type: application/json

{
  "age": 31,
  "status": "premium"
}

Response:
{
  "id": "john-doe",
  "data": {
    "name": "John Doe",
    "email": "john@example.com",
    "age": 31,
    "status": "premium"
  },
  "updated_at": "2024-01-15T10:30:00Z",
  "message": "Document updated successfully"
}

# Delete document
DELETE /collections/{name}/{id}
Authorization: Bearer <token>

Response:
{
  "deleted": true,
  "message": "Document deleted successfully"
}

```

Query Endpoints

```

# Find documents by query
POST /collections/{name}/find
Authorization: Bearer <token>
Content-Type: application/json

{
  "query": {
    "age": {"$gt": 25},
    "status": "active"
  },
  "limit": 10,
  "offset": 0,
  "sort": {"name": 1}
}

Response:
{
  "documents": [

```

```

    {
      "id": "john-doe",
      "data": {
        "name": "John Doe",
        "email": "john@example.com",
        "age": 30,
        "status": "active"
      }
    }
  ],
  "total": 45,
  "limit": 10,
  "offset": 0
}

```

Find one document
 POST /collections/{name}/findone
 Authorization: Bearer <token>
 Content-Type: application/json

```

{
  "query": {
    "email": "john@example.com"
  }
}

```

Response:

```

{
  "id": "john-doe",
  "data": {
    "name": "John Doe",
    "email": "john@example.com",
    "age": 30
  }
}

```

Count documents
 GET /collections/{name}/count
 Authorization: Bearer <token>

Query Parameters:
 - query: JSON query string (URL encoded)

Example:

GET /collections/users/count?query=%7B%22status%22%3A%22active%22%7D

Response:

```

{
  "count": 45,
  "collection": "users"
}

```

Distinct values
 GET /collections/{name}/distinct/{field}
 Authorization: Bearer <token>

Query Parameters:
 - query: JSON query string (URL encoded)

Example:

GET /collections/users/distinct/status?query=%7B%22age%22%3A%7B%22%24gt%22%3A25%7D%7D

Response:

```

{
  "field": "status",
  "values": ["active", "inactive", "premium"],
}

```

```
"count": 3
}
```

Batch Operations Endpoints

```
# Batch insert
POST /collections/{name}/batch
Authorization: Bearer <token>
Content-Type: application/json

{
  "documents": [
    {
      "name": "Alice Johnson",
      "email": "alice@example.com",
      "age": 28
    },
    {
      "name": "Bob Smith",
      "email": "bob@example.com",
      "age": 32
    }
  ]
}

Response:
{
  "inserted": 2,
  "ids": ["alice-johnson", "bob-smith"],
  "message": "Batch insert completed"
}

# Batch update
PUT /collections/{name}/batch
Authorization: Bearer <token>
Content-Type: application/json

{
  "query": {
    "status": "inactive"
  },
  "update": {
    "status": "active",
    "updated_at": "2024-01-15T10:30:00Z"
  }
}

Response:
{
  "updated": 15,
  "message": "Batch update completed"
}

# Batch delete
DELETE /collections/{name}/batch
Authorization: Bearer <token>
Content-Type: application/json

{
  "query": {
    "status": "deleted"
  }
}
```

```

}

Response:
{
  "deleted": 8,
  "message": "Batch delete completed"
}

```

System Endpoints

```

# Health check
GET /health

Response:
{
  "status": "healthy",
  "timestamp": "2024-01-15T10:30:00Z",
  "version": "1.0.0",
  "uptime": 3600
}

# Server info
GET /info

Response:
{
  "name": "GitDB Server",
  "version": "1.0.0",
  "node_version": "18.0.0",
  "platform": "linux",
  "memory_usage": {
    "rss": 52428800,
    "heapTotal": 20971520,
    "heapUsed": 10485760
  }
}

# Server stats
GET /stats

Response:
{
  "requests": {
    "total": 1500,
    "per_minute": 25,
    "errors": 5
  },
  "collections": {
    "total": 5,
    "documents": 1250
  },
  "performance": {
    "avg_response_time": 150,
    "slowest_query": 500
  }
}

```

GraphQL API Reference

GraphQL Endpoint

```
http://localhost:7896/graphql
```

GraphQL Playground

```
http://localhost:7896/graphql-playground
```

Complete Schema

```
scalar JSON
scalar DateTime

type Document {
  id: ID!
  data: JSON!
  created_at: DateTime!
  updated_at: DateTime!
}

type Collection {
  name: String!
  documents: [Document!]!
  count: Int!
}

type QueryResult {
  documents: [Document!]!
  total: Int!
  limit: Int!
  offset: Int!
}

type MutationResult {
  success: Boolean!
  message: String!
  id: ID
  count: Int
}

type Query {
  # Collection queries
  collections: [Collection!]!
  collection(name: String!): Collection

  # Document queries
  document(collection: String!, id: ID!): Document
  documents(
    collection: String!
    limit: Int = 50
    offset: Int = 0
    sort: String
    order: String = "asc"
  ): QueryResult!

  # Search queries
  find(
```

```

    collection: String!
    query: JSON
    limit: Int = 50
    offset: Int = 0
    sort: JSON
  ): QueryResult!

  findOne(
    collection: String!
    query: JSON
  ): Document

  # Aggregation queries
  count(
    collection: String!
    query: JSON
  ): Int!

  distinct(
    collection: String!
    field: String!
    query: JSON
  ): [JSON!]!

  # System queries
  health: JSON!
  info: JSON!
  stats: JSON!
}

type Mutation {
  # Collection mutations
  createCollection(name: String!): MutationResult!
  deleteCollection(name: String!): MutationResult!

  # Document mutations
  insert(
    collection: String!
    document: JSON!
  ): Document!

  update(
    collection: String!
    id: ID!
    document: JSON!
  ): Document!

  delete(
    collection: String!
    id: ID!
  ): MutationResult!

  # Batch mutations
  insertMany(
    collection: String!
    documents: [JSON!]!
  ): [Document!]!

  updateMany(
    collection: String!
    query: JSON!
    update: JSON!
  ): MutationResult!

  deleteMany(
    collection: String!

```

```

        query: JSON!
      ): MutationResult!
    }

    type Subscription {
      documentChanged(collection: String!): Document!
      collectionChanged: Collection!
    }

```

GraphQL Examples

```

# Get all collections
query GetCollections {
  collections {
    name
    count
    created_at
  }
}

# Get documents from collection
query GetUsers($limit: Int, $offset: Int) {
  documents(collection: "users", limit: $limit, offset: $offset) {
    documents {
      id
      data
      created_at
      updated_at
    }
    total
    limit
    offset
  }
}

# Find documents by query
query FindActiveUsers {
  find(collection: "users", query: { status: "active" }) {
    documents {
      id
      data
    }
    total
  }
}

# Find one document
query FindUserByEmail($email: String!) {
  findOne(collection: "users", query: { email: $email }) {
    id
    data
  }
}

# Count documents
query CountActiveUsers {
  count(collection: "users", query: { status: "active" })
}

# Insert document
mutation CreateUser($userData: JSON!) {
  insert(collection: "users", document: $userData) {

```



```

      id
      data
      created_at
    }
  }

  # Update document
  mutation UpdateUser($id: ID!, $updateData: JSON!) {
    update(collection: "users", id: $id, document: $updateData) {
      id
      data
      updated_at
    }
  }

  # Delete document
  mutation DeleteUser($id: ID!) {
    delete(collection: "users", id: $id) {
      success
      message
    }
  }

  # Batch operations
  mutation BatchInsertUsers($users: [JSON!]!) {
    insertMany(collection: "users", documents: $users) {
      id
      data
    }
  }

  # Complex queries
  query ComplexUserQuery {
    find(
      collection: "users"
      query: {
        age: { _gt: 25, _lt: 50 }
        status: "active"
        tags: { _in: ["premium", "vip"] }
      }
      limit: 10
      sort: { name: 1 }
    ) {
      documents {
        id
        data
      }
      total
    }
  }
}

```

JavaScript Client Examples

```

// REST API Client
class GitDBRestClient {
  constructor(baseUrl, token) {
    this.baseUrl = baseUrl;
    this.token = token;
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseUrl}${endpoint}`;
  }
}

```

```

const config = {
  headers: {
    'Authorization': `Bearer ${this.token}`,
    'Content-Type': 'application/json',
    ...options.headers
  },
  ...options
};

const response = await fetch(url, config);
if (!response.ok) {
  throw new Error(`HTTP ${response.status}: ${response.statusText}`);
}
return response.json();
}

// Collection operations
async getCollections() {
  return this.request('/collections');
}

async createCollection(name) {
  return this.request('/collections', {
    method: 'POST',
    body: JSON.stringify({ name })
  });
}

// Document operations
async getDocuments(collection, params = {}) {
  const query = new URLSearchParams(params).toString();
  return this.request(`/collections/${collection}?${query}`);
}

async insertDocument(collection, document) {
  return this.request(`/collections/${collection}`, {
    method: 'POST',
    body: JSON.stringify(document)
  });
}

async getDocument(collection, id) {
  return this.request(`/collections/${collection}/${id}`);
}

async updateDocument(collection, id, update) {
  return this.request(`/collections/${collection}/${id}`, {
    method: 'PUT',
    body: JSON.stringify(update)
  });
}

async deleteDocument(collection, id) {
  return this.request(`/collections/${collection}/${id}`, {
    method: 'DELETE'
  });
}

// Query operations
async find(collection, query, params = {}) {
  return this.request(`/collections/${collection}/find`, {
    method: 'POST',
    body: JSON.stringify({ query, ...params })
  });
}

```

```

async findOne(collection, query) {
  return this.request(`/collections/${collection}/findone`, {
    method: 'POST',
    body: JSON.stringify({ query })
  });
}

async count(collection, query = null) {
  const params = query ? { query: JSON.stringify(query) } : {};
  const queryString = new URLSearchParams(params).toString();
  return this.request(`/collections/${collection}/count?${queryString}`);
}
}

// Usage example
const client = new GitDBRestClient('http://localhost:7896/api/v1', 'ghp_your_token');

// Insert user
const user = await client.insertDocument('users', {
  name: 'John Doe',
  email: 'john@example.com',
  age: 30
});

// Find users
const activeUsers = await client.find('users', { status: 'active' });

// Update user
await client.updateDocument('users', user.id, { age: 31 });

```

```

// GraphQL Client
class GitDBGraphQLClient {
  constructor(endpoint, token) {
    this.endpoint = endpoint;
    this.token = token;
  }

  async query(query, variables = {}) {
    const response = await fetch(this.endpoint, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${this.token}`
      },
      body: JSON.stringify({
        query,
        variables
      })
    });

    const result = await response.json();
    if (result.errors) {
      throw new Error(result.errors[0].message);
    }
    return result.data;
  }

  // Collection operations
  async getCollections() {
    const query = `
      query {
        collections {
          name

```

```

        count
        created_at
    }
}
`;
return this.query(query);
}

// Document operations
async getDocuments(collection, limit = 50, offset = 0) {
    const query = `
        query GetDocuments($collection: String!, $limit: Int, $offset: Int) {
            documents(collection: $collection, limit: $limit, offset: $offset) {
                documents {
                    id
                    data
                    created_at
                    updated_at
                }
                total
                limit
                offset
            }
        }
    `;
    return this.query(query, { collection, limit, offset });
}

async insertDocument(collection, document) {
    const query = `
        mutation InsertDocument($collection: String!, $document: JSON!) {
            insert(collection: $collection, document: $document) {
                id
                data
                created_at
            }
        }
    `;
    return this.query(query, { collection, document });
}

async findDocuments(collection, query, limit = 50, offset = 0) {
    const graphqlQuery = `
        query FindDocuments($collection: String!, $query: JSON, $limit: Int, $offset: Int) {
            find(collection: $collection, query: $query, limit: $limit, offset: $offset) {
                documents {
                    id
                    data
                }
                total
            }
        }
    `;
    return this.query(graphqlQuery, { collection, query, limit, offset });
}
}

// Usage example
const gqlClient = new GitDBGraphQLClient('http://localhost:7896/graphql', 'ghp_your_token')

// Insert user
const result = await gqlClient.insertDocument('users', {
    name: 'John Doe',
    email: 'john@example.com',
    age: 30
});

```

```
// Find active users
const activeUsers = await gqlClient.findDocuments('users', { status: 'active' });
```

Server Management and Monitoring

Process Management

```
# Start server in background
nohup gitdb server > server.log 2>&1 &

# Check if server is running
ps aux | grep gitdb

# Stop server
pkill -f "gitdb server"

# Restart server
pkill -f "gitdb server" && gitdb server
```

Logging and Monitoring

```
# View server logs
tail -f server.log

# Monitor server performance
watch -n 5 'curl -s http://localhost:7896/stats | jq'

# Check server health
curl -s http://localhost:7896/health | jq

# Monitor memory usage
watch -n 5 'curl -s http://localhost:7896/info | jq .memory_usage'
```

Load Balancing

```
# Nginx configuration for load balancing
upstream gitdb_servers {
    server localhost:7896;
    server localhost:7897;
    server localhost:7898;
}

server {
    listen 80;
    server_name api.gitdb.com;

    location / {
        proxy_pass http://gitdb_servers;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

```

    }
  }
}

```

Advanced Server Features {#advanced-server}

WebSocket Support for Real-time Updates

```

// WebSocket connection for real-time updates
const ws = new WebSocket('ws://localhost:7896/ws');

ws.onopen = () => {
  console.log('Connected to GitDB WebSocket');

  // Subscribe to collection changes
  ws.send(JSON.stringify({
    type: 'subscribe',
    collection: 'users'
  }));
};

ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  switch (data.type) {
    case 'document_changed':
      console.log('Document changed:', data.document);
      break;
    case 'collection_changed':
      console.log('Collection changed:', data.collection);
      break;
    case 'error':
      console.error('WebSocket error:', data.error);
      break;
  }
};

ws.onclose = () => {
  console.log('Disconnected from GitDB WebSocket');
};

```

Server Clustering

```

// Cluster configuration
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
}

```

```
cluster.on('exit', (worker, code, signal) => {
  console.log(`Worker ${worker.process.pid} died`);
  cluster.fork(); // Replace dead worker
});
} else {
  // Worker process
  require('./server.js');
  console.log(`Worker ${process.pid} started`);
}
```

Custom Extensions {#custom-extensions}

Building Custom GitDB Extensions

GitDB supports custom extensions for specialized functionality:

Extension Structure

```
// my-extension.js
class MyGitDBExtension {
  constructor(client) {
    this.client = client;
  }

  async customMethod(collection, data) {
    // Custom logic here
    const result = await this.client.insert(collection, data);
    return this.processResult(result);
  }

  processResult(result) {
    // Custom processing
    return {
      ...result,
      processed: true,
      timestamp: new Date().toISOString()
    };
  }
}

module.exports = MyGitDBExtension;
```

Using Extensions

```
const GitDBClient = require('gitdb-client');
const MyExtension = require('./my-extension');

const client = new GitDBClient({
  owner: 'yourusername',
  repo: 'my-database',
  token: 'your-token'
```

```
});  
  
// Add extension  
client.use(new MyExtension(client));  
  
// Use custom method  
const result = await client.customMethod('users', { name: 'John' });
```

Conclusion

GitDB represents a paradigm shift in database technology, combining the reliability and versioning capabilities of Git with the flexibility and power of modern document databases. Whether you're building a small application or a large-scale system, GitDB provides the tools and features you need to succeed.

Key Takeaways

GitDB is more than just a database - it's a complete data management solution

Version control is built-in - every change is tracked and can be reverted

Distributed by design - no central server required

Developer-friendly - familiar Git workflow with powerful SDKs

Enterprise-ready - scales from small projects to large applications

Cost-effective - no infrastructure costs, uses existing Git hosting

Getting Started

Create GitHub Repository: Go to GitHub.com and create a new private repository

Generate GitHub Token: Create a personal access token with `repo` permissions

Install GitDB CLI: `npm install -g gitdb-database`

Connect to Database: `gitdb connect -t <token> -o <owner> -r <repo>`

Choose your SDK: JavaScript, Python, Go, Rust, PHP, or Ruby

Start building: Use the examples and best practices in this guide

Community & Support

- GitHub: <https://github.com/youru/gitdb>
- Issues: <https://github.com/yourusername/gitdb/issues>

Contributing

GitDB is open source and welcomes contributions:

- Fork the repository
- Create a feature branch
- Make your changes
- Submit a pull request

GitDB Where data meets version control.

This book covers everything you need to know about GitDB. From basic concepts to advanced features, real-world examples to performance optimization, you now have the complete guide to building powerful applications with GitDB.

Last updated: July 2025

Version: 2.2.1

GitDB Where data meets version control

Generated on 14/7/2025