Redis OM Node.js

❖ Introduction

- Redis OM Node.js is a TypeScript-first library that provides object mapping and fluent querying capabilities for Redis Stack. It offers a modern, async/await-based API that feels natural to JavaScript and TypeScript developers.
- **Key Features:**
  - TypeScript-first design with full type safety
  - Entity and Schema-based object mapping
  - Fluent query API with method chaining
  - Support for both Redis Hashes and JSON documents
  - Automatic index creation and management
  - Repository pattern for CRUD operations
  - Integration with popular Node.js frameworks
- **Architecture Components:**
  - **Entity:** Base class for domain objects
  - **Schema:** Defines field types and indexing behavior
  - **Client:** Manages Redis connections
  - **Repository:** Provides CRUD and query operations

❖ Installation and Setup

- **Prerequisites**
  - Node.js 14+ (recommended: Node.js 18+)
  - Redis Stack (Redis with RedisJSON and RediSearch modules)
  - TypeScript (for type safety, optional but recommended)

- **Installation**

```
# Install Redis OM Node.js
```

```
npm install redis-om

# Install TypeScript and type definitions (optional)
npm install -D typescript @types/node

# Install Express for web framework integration
npm install express
npm install -D @types/express
```

➢ Redis Stack Setup

```
# Using Docker
docker run -p 6379:6379 -p 8001:8001 redis/redis-stack

# Using Docker Compose
version: '3.8'
services:
  redis-stack:
    image: redis/redis-stack
    ports:
      - "6379:6379"
      - "8001:8001"
```

➢ Basic Client Setup

```
import { Client } from 'redis-om';

// Create and connect to Redis
const client = new Client();

async function connectToRedis() {
    try {
        // Connect to Redis Stack
        await client.open('redis://localhost:6379');
        console.log('Connected to Redis Stack');
    } catch (error) {
        console.error('Failed to connect to Redis:', error);
        process.exit(1);
    }
}

// Environment-based configuration
const redisUrl = process.env.REDIS_URL ||
'redis://localhost:6379';
await client.open(redisUrl);
```

➢ Advanced Client Configuration

```
import { Client } from 'redis-om';
```

```
const client = new Client({
    host: process.env.REDIS_HOST || 'localhost',
    port: parseInt(process.env.REDIS_PORT || '6379'),
    password: process.env.REDIS_PASSWORD,
    username: process.env.REDIS_USERNAME || 'default',
    database: parseInt(process.env.REDIS_DATABASE || '0'),

    // SSL configuration
    tls: process.env.REDIS_SSL === 'true' ? {} : undefined,

    // Connection pool settings
    socket: {
        reconnectStrategy: (retries) => Math.min(retries * 50,
500),
        connectTimeout: 10000,
        commandTimeout: 5000
    }
});

await client.open();
```

❖ Entity Schema Definitions

➢ Basic Entity and Schema

```
import { Entity, Schema, Repository } from 'redis-om';

// Define Entity class
class Album extends Entity {
    // Properties are defined in the schema, not here
    // but you can add custom methods

    get fullTitle(): string {
        return `${this.artist} - ${this.title}`;
    }

    isClassicRock(): boolean {
        return this.genres?.includes('classic rock') || false;
    }
}

// Define Schema
const albumSchema = new Schema(Album, {
    artist: { type: 'string' },              // Tag index for
exact matching
    title: { type: 'string', textSearch: true }, // Full-text
search
    year: { type: 'number' },                // Numeric index
```

```
    genres: { type: 'string[]' },            // Array of
strings (tag index)
    duration: { type: 'number' },            // Duration in
seconds
    isAvailable: { type: 'boolean' },        // Boolean field
    releaseDate: { type: 'date' },           // Date field
    metadata: { type: 'object' }             // JSON object
(stored as-is)
}, {
    dataStructure: 'JSON',                   // Use JSON
documents (default: HASH)
    prefix: 'album'                          // Key prefix for
Redis keys
});
```

➢ Advanced Schema Configuration

```
import { Entity, Schema } from 'redis-om';

class Product extends Entity {}

const productSchema = new Schema(Product, {
    // String fields
    name: {
        type: 'string',
        textSearch: true,           // Enable full-text search
        weight: 2.0,               // Boost search relevance
        sortable: true             // Enable sorting
    },

    category: {
        type: 'string',
        caseSensitive: false       // Case-insensitive
matching
    },

    // Numeric fields
    price: {
        type: 'number',
        sortable: true
    },

    rating: {
        type: 'number',
        min: 0,                    // Validation (not enforced
by Redis)
        max: 5
    },

    // Array fields
    tags: {
```

```
        type: 'string[]',
        separator: '|'              // Custom separator for
storage
    },

    // Boolean fields
    inStock: { type: 'boolean' },
    featured: { type: 'boolean' },

    // Date fields
    createdAt: { type: 'date' },
    updatedAt: { type: 'date' },

    // Point (geospatial) fields
    location: { type: 'point' },

    // Text fields (full-text search)
    description: {
        type: 'text',
        textSearch: true,
        stemming: true,             // Enable stemming
        language: 'english'         // Language for stemming
    },

    // Object fields (JSON)
    specifications: { type: 'object' },

    // Vector fields (for similarity search)
    embedding: {
        type: 'vector',
        dimensions: 1536,           // Vector dimensions
        algorithm: 'FLAT',          // or 'HNSW'
        distance: 'COSINE'          // Distance metric
    }
}, {
    dataStructure: 'JSON',
    prefix: 'product',
    stopWords: ['the', 'a', 'an']  // Custom stop words for
search
});
```

- **TypeScript Interface Integration**

```
// Define TypeScript interface for type safety
interface IUser {
    id?: string;
    email: string;
    firstName: string;
    lastName: string;
    age: number;
    bio?: string;
```

```
    location?: { longitude: number; latitude: number };
    interests: string[];
    isActive: boolean;
    joinedAt: Date;
    preferences: {
        notifications: boolean;
        theme: 'light' | 'dark';
        language: string;
    };
}

// Entity class with interface
class User extends Entity implements IUser {
    // TypeScript will enforce interface compliance
    id?: string;
    email!: string;
    firstName!: string;
    lastName!: string;
    age!: number;
    bio?: string;
    location?: { longitude: number; latitude: number };
    interests!: string[];
    isActive!: boolean;
    joinedAt!: Date;
    preferences!: {
        notifications: boolean;
        theme: 'light' | 'dark';
        language: string;
    };

    // Custom methods
    get fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }

    get isAdult(): boolean {
        return this.age >= 18;
    }
}

const userSchema = new Schema(User, {
    email: { type: 'string', textSearch: true },
    firstName: { type: 'string', sortable: true },
    lastName: { type: 'string', sortable: true },
    age: { type: 'number' },
    bio: { type: 'text', textSearch: true },
    location: { type: 'point' },
    interests: { type: 'string[]' },
    isActive: { type: 'boolean' },
    joinedAt: { type: 'date' },
    preferences: { type: 'object' }
```

```
}, {
    dataStructure: 'JSON',
    prefix: 'user'
});
```

❖ Saving and Querying Data

➢ Repository Creation and CRUD Operations

```
import { Repository } from 'redis-om';

// Create repository instance
const userRepository = new Repository(userSchema, client);

// Ensure indexes are created
await userRepository.createIndex();

async function demonstrateCrudOperations() {
    // CREATE - Single entity
    const user = userRepository.createEntity({
        email: 'john.doe@example.com',
        firstName: 'John',
        lastName: 'Doe',
        age: 30,
        bio: 'Software developer passionate about Redis and
Node.js',
        location: { longitude: -122.4194, latitude: 37.7749 },
// San Francisco
        interests: ['programming', 'databases',
'performance'],
        isActive: true,
        joinedAt: new Date(),
        preferences: {
            notifications: true,
            theme: 'dark',
            language: 'en'
        }
    });

    // Save and get the generated ID
    const userId = await userRepository.save(user);
    console.log('Saved user with ID:', userId);

    // CREATE - Multiple entities
    const users = [
        userRepository.createEntity({
            email: 'alice@example.com',
            firstName: 'Alice',
            lastName: 'Smith',
            age: 28,
```

```
            bio: 'Data scientist specializing in machine
learning',
            interests: ['data-science', 'python', 'ai'],
            isActive: true,
            joinedAt: new Date(),
            preferences: { notifications: false, theme:
'light', language: 'en' }
        }),
        userRepository.createEntity({
            email: 'bob@example.com',
            firstName: 'Bob',
            lastName: 'Johnson',
            age: 35,
            bio: 'Product manager with technical background',
            interests: ['product-management', 'strategy',
'leadership'],
            isActive: true,
            joinedAt: new Date(),
            preferences: { notifications: true, theme: 'dark',
language: 'en' }
        })
    ];

    // Bulk save
    const ids = await Promise.all(users.map(user =>
userRepository.save(user)));
    console.log('Saved users with IDs:', ids);

    // READ - Single entity
    const foundUser = await userRepository.fetch(userId);
    console.log('Found user:', foundUser.fullName);

    // READ - Multiple entities
    const allUsers = await userRepository.fetch(...ids);
    console.log('Found users:', allUsers.map(u =>
u.fullName));

    // UPDATE
    foundUser.age = 31;
    foundUser.bio = 'Senior software developer with Redis
expertise';
    await userRepository.save(foundUser);

    // DELETE - Single entity
    await userRepository.remove(userId);

    // DELETE - Multiple entities
    await userRepository.remove(...ids);

    // DELETE - All entities
```

```
    // await userRepository.dropIndex(); // Optional: drop the
search index
    // const allIds = await
userRepository.search().returnAllIds();
    // await userRepository.remove(...allIds);
}
```

➢ Query Operations

```
async function demonstrateQuerying() {
    // Simple equality search
    const johnUsers = await userRepository
        .search()
        .where('firstName').equals('John')
        .returnAll();

    // Multiple conditions (AND)
    const activeAdults = await userRepository
        .search()
        .where('isActive').equals(true)
        .and('age').greaterThanOrEqualTo(18)
        .returnAll();

    // Numeric range queries
    const youngAdults = await userRepository
        .search()
        .where('age').between(18, 30)
        .returnAll();

    const seniors = await userRepository
        .search()
        .where('age').greaterThan(65)
        .returnAll();

    // String pattern matching
    const techUsers = await userRepository
        .search()
        .where('interests').contains('programming')
        .returnAll();

    // Multiple values (OR condition for same field)
    const techOrDataUsers = await userRepository
        .search()
        .where('interests').containsOneOf('programming',
'data-science', 'ai')
        .returnAll();

    // Full-text search
    const developers = await userRepository
        .search()
        .where('bio').matches('developer')
```

```
        .returnAll();

    // Fuzzy search with wildcards
    const programmers = await userRepository
        .search()
        .where('bio').matches('program*')
        .returnAll();

    // Boolean queries
    const activeUsers = await userRepository
        .search()
        .where('isActive').equals(true)
        .returnAll();

    // Date queries
    const recentUsers = await userRepository
        .search()
        .where('joinedAt').after(new Date('2023-01-01'))
        .returnAll();

    // Complex nested object queries
    const darkThemeUsers = await userRepository
        .search()
        .where('preferences.theme').equals('dark')
        .returnAll();
}
```

➢ Advanced Query Features

```
async function demonstrateAdvancedQuerying() {
    // Geospatial queries (requires point field)
    const nearbyUsers = await userRepository
        .search()
        .where('location')
        .inRadius(
            circle => circle
                .longitude(-122.4194)
                .latitude(37.7749)
                .radius(10)
                .unit('mi')
        )
        .returnAll();

    // Sorting
    const sortedUsers = await userRepository
        .search()
        .where('age').greaterThan(18)
        .sortBy('lastName')
        .returnAll();

    // Multiple sort criteria
```

```
    const complexSort = await userRepository
        .search()
        .where('isActive').equals(true)
        .sortBy('age', 'DESC')
        .sortBy('lastName', 'ASC')
        .returnAll();

    // Pagination
    const firstPage = await userRepository
        .search()
        .where('isActive').equals(true)
        .sortBy('joinedAt', 'DESC')
        .page(0, 10) // offset, count
        .returnAll();

    // Return only IDs (for performance)
    const userIds = await userRepository
        .search()
        .where('age').between(25, 35)
        .returnAllIds();

    // Count results without fetching
    const count = await userRepository
        .search()
        .where('interests').contains('programming')
        .count();

    // Return first result only
    const firstDeveloper = await userRepository
        .search()
        .where('bio').matches('developer')
        .returnFirst();

    // Complex query with multiple conditions
    const specificUsers = await userRepository
        .search()
        .where('age').between(25, 40)
        .and('isActive').equals(true)
        .and('interests').contains('programming')
        .and('preferences.theme').equals('dark')
        .and('bio').matches('senior*')
        .sortBy('joinedAt', 'DESC')
        .page(0, 5)
        .returnAll();

    // Raw Redis query (escape hatch)
    const rawResults = await userRepository
        .search()
        .rawSearch('@age:[25 40] @interests:{programming}
@bio:senior*')
        .returnAll();
```

```
}
```

➢ Vector Similarity Search

```
// Schema with vector field
const documentSchema = new Schema(Document, {
    title: { type: 'string', textSearch: true },
    content: { type: 'text', textSearch: true },
    embedding: {
        type: 'vector',
        dimensions: 1536,
        algorithm: 'FLAT',
        distance: 'COSINE'
    },
    tags: { type: 'string[]' }
}, {
    dataStructure: 'JSON',
    prefix: 'doc'
});

const documentRepository = new Repository(documentSchema,
client);

async function demonstrateVectorSearch() {
    // Create document with embedding
    const doc = documentRepository.createEntity({
        title: 'Redis OM Guide',
        content: 'Comprehensive guide to Redis Object
Mapping',
        embedding: new Float32Array(1536).fill(0.1), //
Example embedding
        tags: ['redis', 'database', 'node.js']
    });

    await documentRepository.save(doc);

    // Vector similarity search
    const queryVector = new Float32Array(1536).fill(0.12);
    const similarDocs = await documentRepository
        .search()
        .where('embedding')
        .inRadius(
            circle => circle
                .vector(queryVector)
                .radius(0.8) // similarity threshold
        )
        .returnAll();

    // Hybrid search (vector + traditional filters)
    const hybridResults = await documentRepository
        .search()
```

```
        .where('tags').contains('redis')
        .and('embedding')
        .inRadius(
            circle => circle
                .vector(queryVector)
                .radius(0.7)
        )
        .returnAll();
}
```

❖Integration with Express

➢ Basic Express Setup

```
import express from 'express';
import { Client, Repository } from 'redis-om';
import { userSchema, User } from './models/User';

const app = express();
app.use(express.json());

// Redis client setup
const client = new Client();
await client.open(process.env.REDIS_URL ||
'redis://localhost:6379');

// Repository setup
const userRepository = new Repository(userSchema, client);
await userRepository.createIndex();

// Middleware to inject repository
app.use((req, res, next) => {
    req.userRepository = userRepository;
    next();
});

// Type augmentation for Express Request
declare global {
    namespace Express {
        interface Request {
            userRepository: Repository<User>;
        }
    }
}
```

➢ REST API Endpoints

```
// GET /users - List all users with pagination
app.get('/users', async (req, res) => {
    try {
```

```
        const page = parseInt(req.query.page as string) || 0;
        const limit = parseInt(req.query.limit as string) ||
10;
        const sortBy = req.query.sortBy as string ||
'joinedAt';
        const sortOrder = req.query.sortOrder as string ||
'DESC';

        let search = req.userRepository.search();

        // Add filters if provided
        if (req.query.isActive !== undefined) {
            search =
search.where('isActive').equals(req.query.isActive ===
'true');
        }

        if (req.query.minAge) {
            search =
search.where('age').greaterThanOrEqualTo(parseInt(req.query.mi
nAge as string));
        }

        if (req.query.maxAge) {
            search =
search.where('age').lessThanOrEqualTo(parseInt(req.query.maxAg
e as string));
        }

        if (req.query.interest) {
            search =
search.where('interests').contains(req.query.interest as
string);
        }

        // Apply sorting and pagination
        const users = await search
            .sortBy(sortBy, sortOrder as 'ASC' | 'DESC')
            .page(page * limit, limit)
            .returnAll();

        // Get total count for pagination info
        const totalCount = await req.userRepository
            .search()
            .count();

        res.json({
            users,
            pagination: {
                page,
                limit,
```

```
                total: totalCount,
                totalPages: Math.ceil(totalCount / limit)
            }
        });
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// GET /users/:id - Get user by ID
app.get('/users/:id', async (req, res) => {
    try {
        const user = await
req.userRepository.fetch(req.params.id);
        if (!user) {
            return res.status(404).json({ error: 'User not
found' });
        }
        res.json(user);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// POST /users - Create new user
app.post('/users', async (req, res) => {
    try {
        // Validate required fields
        const { email, firstName, lastName, age } = req.body;
        if (!email || !firstName || !lastName || !age) {
            return res.status(400).json({
                error: 'Missing required fields: email,
firstName, lastName, age'
            });
        }

        // Check if user already exists
        const existingUsers = await req.userRepository
            .search()
            .where('email').equals(email)
            .returnAll();

        if (existingUsers.length > 0) {
            return res.status(409).json({ error: 'User with
this email already exists' });
        }

        // Create new user
        const user = req.userRepository.createEntity({
            ...req.body,
            joinedAt: new Date(),
```

```
            isActive: req.body.isActive ?? true,
            interests: req.body.interests || [],
            preferences: {
                notifications: true,
                theme: 'light',
                language: 'en',
                ...req.body.preferences
            }
        });

        const userId = await req.userRepository.save(user);
        const savedUser = await
req.userRepository.fetch(userId);

        res.status(201).json(savedUser);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// PUT /users/:id - Update user
app.put('/users/:id', async (req, res) => {
    try {
        const user = await
req.userRepository.fetch(req.params.id);
        if (!user) {
            return res.status(404).json({ error: 'User not
found' });
        }

        // Update fields
        Object.assign(user, req.body);

        await req.userRepository.save(user);
        res.json(user);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// DELETE /users/:id - Delete user
app.delete('/users/:id', async (req, res) => {
    try {
        const user = await
req.userRepository.fetch(req.params.id);
        if (!user) {
            return res.status(404).json({ error: 'User not
found' });
        }

        await req.userRepository.remove(req.params.id);
```

```
        res.status(204).send();
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// GET /users/search - Search users
app.get('/users/search', async (req, res) => {
    try {
        const { q, interests, minAge, maxAge, location, radius
} = req.query;

        let search = req.userRepository.search();

        // Full-text search in bio
        if (q) {
            search = search.where('bio').matches(q as string);
        }

        // Filter by interests
        if (interests) {
            const interestList = (interests as
string).split(',');
            search =
search.where('interests').containsOneOf(...interestList);
        }

        // Age range filter
        if (minAge && maxAge) {
            search = search.where('age').between(
                parseInt(minAge as string),
                parseInt(maxAge as string)
            );
        } else if (minAge) {
            search =
search.where('age').greaterThanOrEqualTo(parseInt(minAge as
string));
        } else if (maxAge) {
            search =
search.where('age').lessThanOrEqualTo(parseInt(maxAge as
string));
        }

        // Geospatial search
        if (location && radius) {
            const [lat, lon] = (location as
string).split(',').map(Number);
            search = search.where('location').inRadius(
                circle => circle
                    .latitude(lat)
                    .longitude(lon)
```

```
                    .radius(parseFloat(radius as string))
                    .unit('mi')
            );
        }

        const users = await search.returnAll();
        res.json({ users, count: users.length });
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// GET /users/stats - Get user statistics
app.get('/users/stats', async (req, res) => {
    try {
        const totalUsers = await
req.userRepository.search().count();
        const activeUsers = await req.userRepository
            .search()
            .where('isActive').equals(true)
            .count();

        // Get age distribution (would require aggregation,
simplified here)
        const allUsers = await
req.userRepository.search().returnAll();
        const ageGroups = allUsers.reduce((acc, user) => {
            const ageGroup = Math.floor(user.age / 10) * 10;
            acc[`${ageGroup}-${ageGroup + 9}`] =
(acc[`${ageGroup}-${ageGroup + 9}`] || 0) + 1;
            return acc;
        }, {} as Record<string, number>);

        // Get popular interests
        const interestCounts = allUsers.reduce((acc, user) =>
{
            user.interests.forEach(interest => {
                acc[interest] = (acc[interest] || 0) + 1;
            });
            return acc;
        }, {} as Record<string, number>);

        const topInterests = Object.entries(interestCounts)
            .sort(([,a], [,b]) => b - a)
            .slice(0, 10)
            .map(([interest, count]) => ({ interest, count
}));

        res.json({
            totalUsers,
            activeUsers,
```

```
            inactiveUsers: totalUsers - activeUsers,
            ageDistribution: ageGroups,
            topInterests
        });
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});
```

➢ Error Handling Middleware

```
// Error handling middleware
app.use((error: Error, req: express.Request, res:
express.Response, next: express.NextFunction) => {
    console.error('API Error:', error);

    if (error.message.includes('Redis')) {
        return res.status(503).json({
            error: 'Database temporarily unavailable',
            type: 'DATABASE_ERROR'
        });
    }

    if (error.message.includes('validation')) {
        return res.status(400).json({
            error: error.message,
            type: 'VALIDATION_ERROR'
        });
    }

    res.status(500).json({
        error: 'Internal server error',
        type: 'INTERNAL_ERROR'
    });
});

// Graceful shutdown
process.on('SIGTERM', async () => {
    console.log('Shutting down gracefully...');
    await client.close();
    process.exit(0);
});

process.on('SIGINT', async () => {
    console.log('Shutting down gracefully...');
    await client.close();
    process.exit(0);
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
```

```
    console.log(`Server running on port ${PORT}`);
});
```

❖Async/Await Handling

➢ Best Practices for Async Operations

```
// Proper error handling with async/await
async function safeRepositoryOperation() {
    try {
        const user = await userRepository.fetch('some-id');
        return user;
    } catch (error) {
        console.error('Repository operation failed:', error);
        throw new Error('Failed to fetch user');
    }
}

// Batch operations with proper error handling
async function batchCreateUsers(userData: Partial<User>[]) {
    const results = [];
    const errors = [];

    for (const data of userData) {
        try {
            const user = userRepository.createEntity(data);
            const id = await userRepository.save(user);
            results.push({ id, success: true });
        } catch (error) {
            errors.push({ data, error: error.message });
        }
    }

    return { results, errors };
}

// Parallel operations with Promise.all
async function fetchMultipleUsers(ids: string[]) {
    try {
        const users = await Promise.all(
            ids.map(id => userRepository.fetch(id))
        );
        return users.filter(user => user !== null);
    } catch (error) {
        console.error('Failed to fetch multiple users:',
error);
        throw error;
    }
}
```

```
// Safe parallel operations with Promise.allSettled
async function safeFetchMultipleUsers(ids: string[]) {
    const results = await Promise.allSettled(
        ids.map(id => userRepository.fetch(id))
    );

    const users = [];
    const errors = [];

    results.forEach((result, index) => {
        if (result.status === 'fulfilled' && result.value) {
            users.push(result.value);
        } else {
            errors.push({ id: ids[index], error: result.reason
});
        }
    });

    return { users, errors };
}
```

➢ Connection Management

```
class RedisConnectionManager {
    private client: Client;
    private repositories: Map<string, Repository<any>>;

    constructor() {
        this.client = new Client();
        this.repositories = new Map();
    }

    async connect(url: string = 'redis://localhost:6379') {
        try {
            await this.client.open(url);
            console.log('Connected to Redis');
        } catch (error) {
            console.error('Failed to connect to Redis:',
error);
            throw error;
        }
    }

    async disconnect() {
        try {
            await this.client.close();
            console.log('Disconnected from Redis');
        } catch (error) {
            console.error('Error disconnecting from Redis:',
error);
        }
```

```
    }

    getRepository<T extends Entity>(schema: Schema<T>, name:
string): Repository<T> {
        if (!this.repositories.has(name)) {
            const repository = new Repository(schema,
this.client);
            this.repositories.set(name, repository);
        }
        return this.repositories.get(name)!;
    }

    async createAllIndexes() {
        const indexPromises =
Array.from(this.repositories.values()).map(
            repo => repo.createIndex().catch(error => {
                console.error('Failed to create index:',
error);
                return null;
            })
        );

        await Promise.allSettled(indexPromises);
    }

    async healthCheck(): Promise<boolean> {
        try {
            await this.client.execute(['PING']);
            return true;
        } catch (error) {
            return false;
        }
    }
}

// Usage example
const connectionManager = new RedisConnectionManager();
await connectionManager.connect(process.env.REDIS_URL);

const userRepo = connectionManager.getRepository(userSchema,
'users');
const productRepo =
connectionManager.getRepository(productSchema, 'products');

await connectionManager.createAllIndexes();
```

❖ Advanced Features

- **Custom Entity Methods and Validation**

```
class User extends Entity {
    // Computed properties
    get fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }

    get displayName(): string {
        return this.fullName || this.email;
    }

    get isAdult(): boolean {
        return this.age >= 18;
    }

    get accountAge(): number {
        return Date.now() - this.joinedAt.getTime();
    }

    // Business logic methods
    updatePreferences(updates: Partial<User['preferences']>) {
        this.preferences = { ...this.preferences, ...updates
};
    }

    addInterest(interest: string) {
        if (!this.interests.includes(interest)) {
            this.interests.push(interest);
        }
    }

    removeInterest(interest: string) {
        this.interests = this.interests.filter(i => i !==
interest);
    }

    activate() {
        this.isActive = true;
    }

    deactivate() {
        this.isActive = false;
    }

    // Validation methods
    validate(): string[] {
        const errors: string[] = [];

        if (!this.email || !this.email.includes('@')) {
            errors.push('Valid email is required');
        }
```

```
        if (!this.firstName || this.firstName.trim().length
=== 0) {
            errors.push('First name is required');
        }

        if (!this.lastName || this.lastName.trim().length ===
0) {
            errors.push('Last name is required');
        }

        if (this.age < 0 || this.age > 150) {
            errors.push('Age must be between 0 and 150');
        }

        if (this.interests.length === 0) {
            errors.push('At least one interest is required');
        }

        return errors;
    }

    isValid(): boolean {
        return this.validate().length === 0;
    }
}
```

➢ Repository Patterns and Service Layer

```
// Repository wrapper with business logic
class UserService {
    constructor(private repository: Repository<User>) {}

    async createUser(userData: Partial<User>): Promise<User> {
        // Validate data
        if (!userData.email || !userData.firstName ||
!userData.lastName) {
            throw new Error('Missing required fields');
        }

        // Check for duplicate email
        const existing = await
this.findByEmail(userData.email);
        if (existing) {
            throw new Error('User with this email already
exists');
        }

        // Create user with defaults
        const user = this.repository.createEntity({
            ...userData,
            joinedAt: new Date(),
```

```
            isActive: true,
            interests: userData.interests || [],
            preferences: {
                notifications: true,
                theme: 'light',
                language: 'en',
                ...userData.preferences
            }
        });

        // Validate before saving
        const validationErrors = user.validate();
        if (validationErrors.length > 0) {
            throw new Error(`Validation failed:
${validationErrors.join(', ')}`);
        }

        const id = await this.repository.save(user);
        return await this.repository.fetch(id);
    }

    async findByEmail(email: string): Promise<User | null> {
        const users = await this.repository
            .search()
            .where('email').equals(email)
            .returnAll();

        return users.length > 0 ? users[0] : null;
    }

    async findActiveUsersByInterest(interest: string):
Promise<User[]> {
        return await this.repository
            .search()
            .where('isActive').equals(true)
            .and('interests').contains(interest)
            .sortBy('joinedAt', 'DESC')
            .returnAll();
    }

    async updateUser(id: string, updates: Partial<User>):
Promise<User> {
        const user = await this.repository.fetch(id);
        if (!user) {
            throw new Error('User not found');
        }

        // Apply updates
        Object.assign(user, updates);

        // Validate
```

```
        const validationErrors = user.validate();
        if (validationErrors.length > 0) {
            throw new Error(`Validation failed:
${validationErrors.join(', ')}`);
        }

        await this.repository.save(user);
        return user;
    }

    async deactivateUser(id: string): Promise<void> {
        const user = await this.repository.fetch(id);
        if (!user) {
            throw new Error('User not found');
        }

        user.deactivate();
        await this.repository.save(user);
    }

    async getUserStats(): Promise<{
        total: number;
        active: number;
        averageAge: number;
        topInterests: Array<{ interest: string; count: number
}>;
    }> {
        const allUsers = await
this.repository.search().returnAll();
        const activeUsers = allUsers.filter(u => u.isActive);

        const averageAge = allUsers.reduce((sum, user) => sum
+ user.age, 0) / allUsers.length;

        const interestCounts = allUsers.reduce((acc, user) =>
{
            user.interests.forEach(interest => {
                acc[interest] = (acc[interest] || 0) + 1;
            });
            return acc;
        }, {} as Record<string, number>);

        const topInterests = Object.entries(interestCounts)
            .sort(([,a], [,b]) => b - a)
            .slice(0, 5)
            .map(([interest, count]) => ({ interest, count
}));

        return {
            total: allUsers.length,
            active: activeUsers.length,
```

```
                averageAge,
                topInterests
            };
        }
    }
```
