# Redis OM Python

## ❖ Introduction

➢ Redis OM Python provides a modern, Pythonic way to interact with Redis Stack using familiar patterns from the Python ecosystem. Built on top of Pydantic, it offers both synchronous and asynchronous support, making it perfect for modern Python applications.

### ➢ Key Features:

- **Pydantic Integration:** Leverage Pydantic models for data validation and serialization
- **Type Safety:** Full type hints and runtime validation
- **Async/Sync Support:** Choose the approach that fits your application
- **FastAPI Integration:** Native support for FastAPI applications
- **Query DSL:** Pythonic query building with method chaining
- **Automatic Indexing:** Schema-based index creation and management
- **Vector Search:** Support for vector similarity search and embeddings

### ➢ Core Components:

- **Model:** Pydantic-based model classes with Redis persistence
- **Field Types:** Rich field types with indexing capabilities
- **Migrator:** Schema migration and index management
- **Query Expressions:** Fluent query building interface

---

## ❖ Setup using pip and Python Environment

### ➢ Prerequisites

- Python 3.7+ (recommended: Python 3.9+)
- Redis Stack (Redis with RedisJSON and RediSearch modules)
- pip or poetry for package management

## ➢ Installation

```
# Install Redis OM Python
pip install redis-om

# For async support
pip install redis-om[async]

# For FastAPI integration
pip install redis-om[fastapi]
pip install fastapi uvicorn

# Development dependencies
pip install pytest pytest-asyncio black isort mypy
```

## ➢ 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"
    environment:
      - REDIS_ARGS=--requirepass yourpassword
```

## ➢ Environment Configuration

```
# config.py
import os
from typing import Optional
from pydantic import BaseSettings

class Settings(BaseSettings):
    redis_url: str = "redis://localhost:6379"
    redis_host: str = "localhost"
    redis_port: int = 6379
    redis_password: Optional[str] = None
    redis_db: int = 0
    redis_ssl: bool = False

    # Application settings
    app_name: str = "Redis OM Python App"
    debug: bool = False
    log_level: str = "INFO"

    class Config:
        env_file = ".env"
        case_sensitive = False

settings = Settings()
```

## ➢ Basic Connection Setup

```
# database.py
import redis.asyncio as redis
from redis.asyncio import Redis
from redis_om import get_redis_connection
from typing import Optional
import asyncio
import logging

logger = logging.getLogger(__name__)

class RedisManager:
    def __init__(self):
        self._redis: Optional[Redis] = None
        self._connection_url = "redis://localhost:6379"

    async def connect(self, url: str = None) -> Redis:
        """Connect to Redis and return connection instance."""
        if url:
            self._connection_url = url

        try:
            self._redis =
get_redis_connection(url=self._connection_url)
            # Test connection
            await self._redis.ping()
            logger.info("Successfully connected to Redis")
            return self._redis
        except Exception as e:
            logger.error(f"Failed to connect to Redis: {e}")
            raise

    async def disconnect(self):
        """Close Redis connection."""
        if self._redis:
            await self._redis.close()
            logger.info("Redis connection closed")

    @property
    def redis(self) -> Redis:
        """Get current Redis connection."""
        if not self._redis:
            raise RuntimeError("Redis not connected. Call
connect() first.")
        return self._redis
    async def health_check(self) -> bool:
        """Check if Redis is healthy."""
        try:
            await self._redis.ping()
            return True
        except Exception:
            return False

# Global instance
redis_manager = RedisManager()
```

---

## ❖ Pydantic Model Integration with Comprehensive Validation

- Redis OM Python is built on top of Pydantic, providing powerful data validation, serialization, and type safety features. This integration ensures data integrity and provides excellent developer experience with full IDE support.
- **Basic Model Definition**

```
# models/base.py
from datetime import datetime
from typing import Optional, List, Union
from pydantic import BaseModel, Field, validator,
root_validator
from redis_om import HashModel, EmbeddedJsonModel
import uuid

class TimestampMixin(BaseModel):
    """Mixin for timestamp fields."""
    created_at: datetime =
Field(default_factory=datetime.utcnow)
    updated_at: Optional[datetime] = None

    class Config:
        json_encoders = {
            datetime: lambda v: v.isoformat()
        }

class BaseRedisModel(HashModel, TimestampMixin):
    """Base model with common functionality."""

    class Meta:
        global_key_prefix = "app"
```

---

## ❖ Integration with FastAPI

➢ Redis OM Python integrates seamlessly with FastAPI, providing automatic API documentation, request/response validation, and efficient async operations.

➢ **FastAPI Application Setup**

```
# main.py
from fastapi import FastAPI, HTTPException, Depends, Query,
Path, Body
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from fastapi.exceptions import RequestValidationError
from contextlib import asynccontextmanager
from typing import List, Optional
import logging
import uvicorn

from database import redis_manager
from services.user_service import user_service
from services.index_manager import index_manager
from api.routes import users, auth, search
from api.exceptions import setup_exception_handlers
from api.middleware import setup_middleware

logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Application lifespan events."""
    # Startup
    logger.info("Starting Redis OM FastAPI application...")

    try:
        # Connect to Redis
        await redis_manager.connect()

        # Create indexes
        await index_manager.create_indexes()

        logger.info("Application startup completed
successfully")
        yield

    except Exception as e:
        logger.error(f"Application startup failed: {e}")
        raise
    finally:
        # Shutdown
        logger.info("Shutting down Redis OM FastAPI
application...")
```

```
        await redis_manager.disconnect()
        logger.info("Application shutdown completed")

# Create FastAPI app
app = FastAPI(
    title="Redis OM Python API",
    description="A comprehensive FastAPI application using
Redis OM Python",
    version="1.0.0",
    docs_url="/docs",
    redoc_url="/redoc",
    lifespan=lifespan
)

# Setup middleware
setup_middleware(app)

# Setup exception handlers
setup_exception_handlers(app)

# Include routers
app.include_router(auth.router, prefix="/api/v1/auth",
tags=["Authentication"])
app.include_router(users.router, prefix="/api/v1/users",
tags=["Users"])
app.include_router(search.router, prefix="/api/v1/search",
tags=["Search"])

@app.get("/")
async def root():
    """Root endpoint."""
    return {"message": "Redis OM Python FastAPI Application",
"status": "running"}

@app.get("/health")
async def health_check():
    """Health check endpoint."""
    redis_healthy = await redis_manager.health_check()

    return {
        "status": "healthy" if redis_healthy else "unhealthy",
        "redis": "connected" if redis_healthy else
"disconnected",
        "timestamp": datetime.utcnow().isoformat()
    }

if __name__ == "__main__":
    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8000,
        reload=True,
        log_level="info"
    )
```

## ➢ FastAPI Routes with Redis OM Models

```
# api/routes/users.py
from fastapi import APIRouter, HTTPException, Depends, Query, 
Path, Body, status
from fastapi.responses import JSONResponse
from typing import List, Optional, Dict, Any
from datetime import datetime

from models.user import User, UserRole, AccountStatus
from services.user_service import user_service
from api.schemas import (
    UserCreate, UserUpdate, UserResponse, UserList,
    PaginationParams, SearchParams
)
from api.dependencies import get_current_user, get_admin_user
from api.exceptions import UserNotFoundError, 
DuplicateUserError

router = APIRouter()

@router.post(
    "/",
    response_model=UserResponse,
    status_code=status.HTTP_201_CREATED,
    summary="Create a new user",
    description="Create a new user with comprehensive 
validation"
)
async def create_user(
    user_data: UserCreate = Body(..., description="User 
creation data")
) -> UserResponse:
    """Create a new user."""
    try:
        # Convert Pydantic model to dict
        user_dict = user_data.dict(exclude_unset=True)

        # Create user through service
        user = await user_service.create_user(user_dict)

        return UserResponse.from_orm(user)

    except ValueError as e:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=str(e)
        )
    except Exception as e:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to create user"
        )

@router.get(
    "/{user_id}",
    response_model=UserResponse,
    summary="Get user by ID",
    description="Retrieve a user by their unique identifier"
)
async def get_user(
    user_id: str = Path(..., description="User ID",
min_length=1),
    current_user: User = Depends(get_current_user)
) -> UserResponse:
    """Get user by ID."""
    user = await user_service.find_by_id(user_id)

    if not user:
        raise UserNotFoundError(user_id)

    # Privacy check
    if user.pk != current_user.pk and
user.preferences.privacy_level == "private":
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="User profile is private"
        )

    return UserResponse.from_orm(user)

@router.get(
    "/",
    response_model=UserList,
    summary="List users with filtering and pagination",
    description="Get a paginated list of users with optional
filtering"
)
async def list_users(
    pagination: PaginationParams = Depends(),
    search: SearchParams = Depends(),
    role: Optional[UserRole] = Query(None, description="Filter
by user role"),
    status: Optional[AccountStatus] = Query(None,
description="Filter by account status"),
    is_active: Optional[bool] = Query(None,
description="Filter by active status"),
    current_user: User = Depends(get_current_user)
```

```
) -> UserList:
    """List users with filtering and pagination."""

    # Build search parameters
    search_params = {
        "limit": pagination.limit,
        "offset": pagination.offset,
        "is_active": is_active
    }

    if role:
        search_params["role"] = role
    if status:
        search_params["status"] = status
    if search.query:
        search_params["bio_search"] = search.query
    if search.interests:
        search_params["interests"] = search.interests

    # Execute search
    users = await user_service.search_users(search_params)
    total_count = await
user_service.count_users(search_params)

    # Filter private profiles
    filtered_users = []
    for user in users:
        if user.preferences.privacy_level != "private" or
user.pk == current_user.pk:
            filtered_users.append(UserResponse.from_orm(user))

    return UserList(
        users=filtered_users,
        total=total_count,
        page=pagination.page,
        per_page=pagination.limit,
        pages=(total_count + pagination.limit - 1) //
pagination.limit
    )

@router.put(
    "/{user_id}",
    response_model=UserResponse,
    summary="Update user",
    description="Update user information with validation"
)
async def update_user(
    user_id: str = Path(..., description="User ID"),
    user_data: UserUpdate = Body(..., description="User update
data"),
    current_user: User = Depends(get_current_user)
```

```
) -> UserResponse:
    """Update user information."""

    # Permission check
    if user_id != current_user.pk and current_user.role not in
[UserRole.ADMIN, UserRole.MODERATOR]:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to update this user"
        )

    # Convert to dict and filter out None values
    updates = user_data.dict(exclude_unset=True,
exclude_none=True)

    try:
        updated_user = await user_service.update_user(user_id,
updates)

        if not updated_user:
            raise UserNotFoundError(user_id)

        return UserResponse.from_orm(updated_user)

    except ValueError as e:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=str(e)
        )

@router.delete(
    "/{user_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    summary="Delete user",
    description="Delete a user account (admin only)"
)
async def delete_user(
    user_id: str = Path(..., description="User ID"),
    current_user: User = Depends(get_admin_user)
):
    """Delete a user account."""

    success = await user_service.delete_user(user_id)

    if not success:
        raise UserNotFoundError(user_id)

    return {"message": "User deleted successfully"}

@router.get(
    "/{user_id}/stats",
    response_model=Dict[str, Any],
    summary="Get user statistics",
    description="Get detailed statistics for a user"
)
async def get_user_stats(
    user_id: str = Path(..., description="User ID"),
    current_user: User = Depends(get_current_user)
) -> Dict[str, Any]:
    """Get user statistics."""

    # Permission check
    if user_id != current_user.pk and current_user.role !=
UserRole.ADMIN:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to view user statistics"
        )

    user = await user_service.find_by_id(user_id)
    if not user:
        raise UserNotFoundError(user_id)

    stats = await user_service.get_user_statistics(user_id)
    return stats

@router.post(
    "/batch",
    response_model=Dict[str, Any],
    summary="Batch create users",
    description="Create multiple users in a single request
(admin only)"
)
async def create_users_batch(
    users_data: List[UserCreate] = Body(..., description="List
of users to create"),
    current_user: User = Depends(get_admin_user)
) -> Dict[str, Any]:
    """Create multiple users in batch."""

    if len(users_data) > 100:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Maximum 100 users allowed per batch"
        )

    # Convert to dicts
    users_dicts = [user.dict(exclude_unset=True) for user in
users_data]

    # Create users
    created_users, errors = await
user_service.create_users_batch(users_dicts)

    return {
        "created_count": len(created_users),
        "error_count": len(errors),
        "created_users": [UserResponse.from_orm(user) for user
in created_users],
        "errors": errors
    }
```

---

## ❖ Indexing and Search Operations

### ➢ Index Management

```
# services/index_manager.py
from redis_om import Migrator
from models.user import User, UserProfile
from models.document import Document
import logging

logger = logging.getLogger(__name__)

class IndexManager:
    """Manages Redis search indexes for all models."""

    def __init__(self):
        self.models = [User, UserProfile, Document]

    async def create_indexes(self):
        """Create all search indexes."""
        try:
            # Run migrations to create indexes
            Migrator().run()
            logger.info("Successfully created all indexes")
        except Exception as e:
            logger.error(f"Failed to create indexes: {e}")
            raise

    async def drop_indexes(self):
        """Drop all search indexes (use with caution)."""
        for model in self.models:
            try:
                await model.db().execute_command(
                    "FT.DROPINDEX",
                    model.Meta.index_name
                )
                logger.info(f"Dropped index for
{model.__name__}")
```

```
            except Exception as e:
                logger.warning(f"Could not drop index for
{model.__name__}: {e}")

    async def get_index_info(self, model_class):
        """Get information about a model's index."""
        try:
            result = await model_class.db().execute_command(
                "FT.INFO",
                model_class.Meta.index_name
            )
            return result
        except Exception as e:
            logger.error(f"Could not get index info for
{model_class.__name__}: {e}")
            return None

    async def rebuild_index(self, model_class):
        """Rebuild index for a specific model."""
        try:
            # Drop existing index
            await model_class.db().execute_command(
                "FT.DROPINDEX",
                model_class.Meta.index_name
            )

            # Recreate index
            Migrator().run()
            logger.info(f"Rebuilt index for
{model_class.__name__}")
        except Exception as e:
            logger.error(f"Failed to rebuild index for
{model_class.__name__}: {e}")
            raise

index_manager = IndexManager()
```

- **Basic Search Operations**

```
# services/user_service.py
from typing import List, Optional, Dict, Any
from datetime import datetime, timedelta
from models.user import User, Address
from redis_om import NotFoundError
import logging

logger = logging.getLogger(__name__)

class UserService:
    """Service class for user operations."""
    async def create_user(self, user_data: Dict[str, Any]) ->
User:
        """Create a new user."""
        try:
            # Check if user already exists
            existing = await
self.find_by_email(user_data.get('email'))
            if existing:
                raise ValueError("User with this email already
exists")

            # Create user instance
            user = User(**user_data)

            # Save to Redis
            await user.save()
            logger.info(f"Created user: {user.pk}")
            return user

        except Exception as e:
            logger.error(f"Failed to create user: {e}")
            raise

    async def find_by_id(self, user_id: str) ->
Optional[User]:
        """Find user by ID."""
        try:
            return await User.get(user_id)
        except NotFoundError:
            return None

    async def find_by_email(self, email: str) ->
Optional[User]:
        """Find user by email address."""
        users = await User.find(User.email == email).all()
        return users[0] if users else None

    async def find_active_users(self, limit: int = 100) ->
List[User]:
        """Find all active users."""
        return await User.find(User.is_active ==
True).limit(limit).all()

    async def search_by_name(self, name_query: str) ->
List[User]:
        """Search users by first or last name."""
        return await User.find(
            (User.first_name % name_query) |
            (User.last_name % name_query)
        ).all()
    async def find_by_age_range(self, min_age: int, max_age:
int) -> List[User]:
        """Find users within age range."""
        return await User.find(
            User.age >= min_age,
            User.age <= max_age
        ).all()

    async def find_by_interests(self, interests: List[str]) ->
List[User]:
        """Find users with specific interests."""
        # Find users who have any of the specified interests
        query = None
        for interest in interests:
            condition = User.interests << interest
            query = condition if query is None else query |
condition

        return await User.find(query).all() if query else []

    async def search_bio(self, search_term: str) ->
List[User]:
        """Full-text search in user bio."""
        return await User.find(User.bio % search_term).all()

    async def find_by_location(self, city: str, state: str =
None) -> List[User]:
        """Find users by location."""
        query = User.address.city == city
        if state:
            query = query & (User.address.state == state)

        return await User.find(query).all()

    async def find_recent_users(self, days: int = 30) ->
List[User]:
        """Find users created in the last N days."""
        cutoff_date = datetime.utcnow() - timedelta(days=days)
        return await User.find(User.created_at >=
cutoff_date).all()

    async def update_user(self, user_id: str, updates:
Dict[str, Any]) -> Optional[User]:
        """Update user data."""
        try:
            user = await self.find_by_id(user_id)
            if not user:
                return None

            # Apply updates
            for key, value in updates.items():
                if hasattr(user, key):
                    setattr(user, key, value)

            user.updated_at = datetime.utcnow()
            await user.save()

            logger.info(f"Updated user: {user_id}")
            return user

        except Exception as e:
            logger.error(f"Failed to update user {user_id}:
{e}")
            raise

    async def delete_user(self, user_id: str) -> bool:
        """Delete a user."""
        try:
            user = await self.find_by_id(user_id)
            if not user:
                return False

            await User.delete(user_id)
            logger.info(f"Deleted user: {user_id}")
            return True

        except Exception as e:
            logger.error(f"Failed to delete user {user_id}:
{e}")
            raise

    async def get_user_stats(self) -> Dict[str, Any]:
        """Get user statistics."""
        try:
            total_users = len(await User.find().all())
            active_users = len(await User.find(User.is_active
== True).all())

            # Get all users for analysis
            all_users = await User.find().all()

            # Age distribution
            age_groups = {}
            for user in all_users:
                age_group = (user.age // 10) * 10
                key = f"{age_group}-{age_group + 9}"
                age_groups[key] = age_groups.get(key, 0) + 1

            # Interest distribution
            interest_counts = {}
            for user in all_users:
                for interest in user.interests:
                    interest_counts[interest] =
interest_counts.get(interest, 0) + 1

            top_interests = sorted(
                interest_counts.items(),
                key=lambda x: x[1],
                reverse=True
            )[:10]

            return {
                "total_users": total_users,
                "active_users": active_users,
                "inactive_users": total_users - active_users,
                "age_distribution": age_groups,
                "top_interests": [{"interest": k, "count": v}
for k, v in top_interests]
            }

        except Exception as e:
            logger.error(f"Failed to get user stats: {e}")
            raise

user_service = UserService()
```

## ❖ Query DSL Usage

### ➢ Advanced Query Building

```
# services/query_examples.py
from typing import List, Optional, Dict, Any
from datetime import datetime, timedelta
from models.user import User
from models.document import Document
from redis_om import NotFoundError

class QueryExamples:
    """Examples of advanced Redis OM queries."""

    async def basic_queries(self):
        """Basic query examples."""

        # Equality queries
        active_users = await User.find(User.is_active ==
True).all()

        # Multiple conditions (AND)
        young_active_users = await User.find(
            User.is_active == True,
            User.age < 30
        ).all()
        # OR conditions
        tech_or_science_users = await User.find(
            (User.interests << "technology") |
            (User.interests << "science")
        ).all()

        # NOT conditions
        non_verified_users = await User.find(
            ~(User.is_verified == True)
        ).all()

        return {
            "active_users": len(active_users),
            "young_active": len(young_active_users),
            "tech_or_science": len(tech_or_science_users),
            "non_verified": len(non_verified_users)
        }

    async def range_queries(self):
        """Range query examples."""

        # Numeric ranges
        middle_aged = await User.find(
            User.age >= 30,
            User.age <= 50
        ).all()

        # Date ranges
        recent_users = await User.find(
            User.created_at >= datetime.utcnow() -
timedelta(days=30)
        ).all()

        # Multiple range conditions
        recent_adults = await User.find(
            User.age >= 18,
            User.created_at >= datetime.utcnow() -
timedelta(days=90)
        ).all()

        return {
            "middle_aged": len(middle_aged),
            "recent_users": len(recent_users),
            "recent_adults": len(recent_adults)
        }

    async def text_search_queries(self):
        """Full-text search examples."""

        # Simple text search
        developers = await User.find(User.bio %
"developer").all()

        # Multiple terms (AND)
        senior_developers = await User.find(
            User.bio % "senior developer"
        ).all()

        # Wildcard search
        programmers = await User.find(User.bio %
"program*").all()

        # Fuzzy search (approximate matching)
        fuzzy_search = await User.find(User.bio %
"%%develop%%").all()

        return {
            "developers": len(developers),
            "senior_developers": len(senior_developers),
            "programmers": len(programmers),
            "fuzzy_matches": len(fuzzy_search)
        }

    async def array_queries(self):
        """Array/list field queries."""

        # Contains specific item
        python_users = await User.find(User.interests <<
"python").all()

        # Contains any of multiple items
        tech_stack_users = await User.find(
            (User.interests << "python") |
            (User.interests << "javascript") |
            (User.interests << "java")
        ).all()

        # Multiple array conditions
        full_stack_devs = await User.find(
            (User.interests << "frontend") &
            (User.interests << "backend")
        ).all()

        return {
            "python_users": len(python_users),
            "tech_stack_users": len(tech_stack_users),
            "full_stack_devs": len(full_stack_devs)
        }

    async def nested_field_queries(self):
        """Nested object field queries."""
        # Query nested fields
        sf_users = await User.find(User.address.city == "San
Francisco").all()

        # Multiple nested conditions
        ca_users = await User.find(
            User.address.state == "CA",
            User.address.country == "US"
        ).all()

        # Nested field with text search
        bay_area = await User.find(
            User.address.city % "San Francisco" |
            User.address.city % "Oakland" |
            User.address.city % "Berkeley"
        ).all()

        return {
            "sf_users": len(sf_users),
            "ca_users": len(ca_users),
            "bay_area": len(bay_area)
        }

    async def sorting_and_pagination(self):
        """Sorting and pagination examples."""

        # Simple sorting
        users_by_age = await User.find().sort_by("age").all()

        # Descending sort
        newest_users = await User.find().sort_by("-
created_at").limit(10).all()

        # Multiple sort criteria
        sorted_users = await User.find(
            User.is_active == True
        ).sort_by("last_name", "first_name").all()

        # Pagination
        page_1 = await User.find().offset(0).limit(10).all()
        page_2 = await User.find().offset(10).limit(10).all()

        # Count without fetching
        total_count = await User.find(User.is_active ==
True).count()

        return {
            "total_active": total_count,
            "page_1_size": len(page_1),
            "page_2_size": len(page_2),
            "newest_user": newest_users[0].full_name if
newest_users else None
        }

    async def aggregation_queries(self):
        """Aggregation and statistics examples."""

        # This is a simplified example - Redis OM Python
doesn't have
        # built-in aggregation, so we fetch and process in
Python
        all_users = await User.find().all()

        # Group by age ranges
        age_groups = {}
        for user in all_users:
            age_range = f"{(user.age // 10) * 10}-{(user.age
// 10) * 10 + 9}"
            age_groups[age_range] = age_groups.get(age_range,
0) + 1

        # Group by location
        location_groups = {}
        for user in all_users:
            if user.address:
                key = f"{user.address.city},
{user.address.state}"
                location_groups[key] =
location_groups.get(key, 0) + 1

        # Interest statistics
        interest_counts = {}
        for user in all_users:
            for interest in user.interests:
                interest_counts[interest] =
interest_counts.get(interest, 0) + 1

        return {
            "age_distribution": age_groups,
            "location_distribution":
dict(list(location_groups.items())[:10]),
            "top_interests": dict(
                sorted(interest_counts.items(), key=lambda x:
x[1], reverse=True)[:10]
            )
        }

    async def complex_queries(self):
        """Complex query combinations."""

        # Complex business logic query
        target_users = await User.find(
            # Active users
            User.is_active == True,
            # Adults
            User.age >= 18,
            # Recently active (has last_login)
            User.last_login.is_not_null(),
            # Tech-interested
            (User.interests << "technology") |
            (User.interests << "programming") |
            (User.interests << "software"),
            # Bio mentions experience
            User.bio % "experience*",
            # Located in major tech cities
            (User.address.city == "San Francisco") |
            (User.address.city == "Seattle") |
            (User.address.city == "New York") |
            (User.address.city == "Austin")
        ).sort_by("-last_login").limit(50).all()

        return {
            "target_users_count": len(target_users),
            "sample_user": target_users[0].full_name if
target_users else None
        }

query_examples = QueryExamples()
```

➢ **Vector Similarity Search**

```
# services/vector_search.py
import numpy as np
from typing import List, Optional, Tuple
from models.document import Document
from redis_om import NotFoundError

class VectorSearchService:
    """Service for vector similarity search operations."""

    def generate_mock_embedding(self, text: str) ->
List[float]:
        """Generate a mock embedding for demonstration."""
        # In practice, you'd use a real embedding model like:
        # - OpenAI embeddings
        # - Sentence transformers
        # - Hugging Face models
        np.random.seed(hash(text) % (2**32))
        return np.random.normal(0, 1, 1536).tolist()

    async def create_document_with_embedding(
        self,
        title: str,
        content: str,
        author: str,
        category: str,
        tags: List[str] = None
    ) -> Document:
        """Create a document with auto-generated embedding."""

        # Generate embedding for the content
        embedding = self.generate_mock_embedding(f"{title}
{content}")

        doc = Document(
            title=title,
            content=content,
            author=author,
            category=category,
            tags=tags or [],
            embedding=embedding
        )

        await doc.save()
        return doc

    async def find_similar_documents(
        self,
        query_text: str,
        k: int = 10,
        threshold: float = 0.7
    ) -> List[Tuple[Document, float]]:
        """Find documents similar to query text."""

        # Generate embedding for query
        query_embedding =
self.generate_mock_embedding(query_text)

        # In practice, you would use Redis vector search:
        # This is a simplified example since Redis OM Python's
        # vector search syntax may vary

        # For now, we'll simulate by fetching all documents
        # and computing similarity in Python
        all_docs = await Document.find().all()

        similarities = []
        for doc in all_docs:
            if doc.embedding:
                # Compute cosine similarity
                similarity =
self._cosine_similarity(query_embedding, doc.embedding)
                if similarity >= threshold:
                    similarities.append((doc, similarity))
        # Sort by similarity (descending)
        similarities.sort(key=lambda x: x[1], reverse=True)
        return similarities[:k]

    async def find_similar_to_document(
        self,
        document_id: str,
        k: int = 5
    ) -> List[Tuple[Document, float]]:
        """Find documents similar to a given document."""

        try:
            source_doc = await Document.get(document_id)
            if not source_doc.embedding:
                raise ValueError("Source document has no
embedding")

            return await self.find_similar_documents(
                source_doc.content, k=k+1  # +1 to exclude
self
            )

        except NotFoundError:
            raise ValueError("Document not found")

    async def hybrid_search(
        self,
        query_text: str,
        category: Optional[str] = None,
        author: Optional[str] = None,
        tags: Optional[List[str]] = None,
        k: int = 10
    ) -> List[Tuple[Document, float]]:
        """Hybrid search combining vector similarity and
filters."""

        # Start with filtered documents
        query = Document.find()

        if category:
            query = query.filter(Document.category ==
category)

        if author:
            query = query.filter(Document.author == author)

        if tags:
            tag_conditions = [Document.tags << tag for tag in
tags]
            combined_condition = tag_conditions[0]
            for condition in tag_conditions[1:]:
                combined_condition = combined_condition |
condition
            query = query.filter(combined_condition)

        filtered_docs = await query.all()

        # Now compute similarity for filtered documents
        query_embedding =
self.generate_mock_embedding(query_text)
        similarities = []

        for doc in filtered_docs:
            if doc.embedding:
                similarity =
self._cosine_similarity(query_embedding, doc.embedding)
                similarities.append((doc, similarity))

        similarities.sort(key=lambda x: x[1], reverse=True)
        return similarities[:k]

    def _cosine_similarity(self, vec1: List[float], vec2:
List[float]) -> float:
        """Compute cosine similarity between two vectors."""
        vec1 = np.array(vec1)
        vec2 = np.array(vec2)

        dot_product = np.dot(vec1, vec2)
        norm1 = np.linalg.norm(vec1)
        norm2 = np.linalg.norm(vec2)

        if norm1 == 0 or norm2 == 0:
            return 0.0

        return float(dot_product / (norm1 * norm2))

vector_search_service = VectorSearchService()
```

---

## ❖ Async/Sync Support

### ➢ Async Service Implementation

```
# services/async_user_service.py
import asyncio
from typing import List, Optional, Dict, Any, Tuple
from datetime import datetime
from models.user import User
from redis_om import NotFoundError
import logging
```

```python
logger = logging.getLogger(__name__)

class AsyncUserService:
    """Async version of user service with batch operations."""

    async def create_users_batch(self, users_data: 
List[Dict[str, Any]]) -> Tuple[List[User], List[str]]:
        """Create multiple users in batch with error 
handling."""
        created_users = []
        errors = []

        # Create tasks for concurrent execution
        tasks = []
        for i, user_data in enumerate(users_data):
            task = self._create_single_user(user_data, i)
            tasks.append(task)

        # Execute all tasks concurrently
        results = await asyncio.gather(*tasks, 
return_exceptions=True)

        for i, result in enumerate(results):
            if isinstance(result, Exception):
                errors.append(f"User {i}: {str(result)}")
            else:
                created_users.append(result)

        return created_users, errors

    async def _create_single_user(self, user_data: Dict[str, 
Any], index: int) -> User:
        """Create a single user with error handling."""
        try:
            # Validate email uniqueness
            existing = await User.find(User.email == 
user_data['email']).first()
            if existing:
                raise ValueError(f"Email {user_data['email']} 
already exists")

            user = User(**user_data)
            await user.save()
            return user

        except Exception as e:
            logger.error(f"Failed to create user {index}: 
{e}")
            raise
    async def find_users_parallel(self, user_ids: List[str]) -
> Dict[str, Optional[User]]:
        """Find multiple users in parallel."""
        tasks = {user_id: self._find_user_safe(user_id) for
user_id in user_ids}
        results = await asyncio.gather(*tasks.values(),
return_exceptions=True)

        return {
            user_id: result if not isinstance(result,
Exception) else None
            for user_id, result in zip(tasks.keys(), results)
        }

    async def _find_user_safe(self, user_id: str) ->
Optional[User]:
        """Find user with exception handling."""
        try:
            return await User.get(user_id)
        except NotFoundError:
            return None
        except Exception as e:
            logger.error(f"Error finding user {user_id}: {e}")
            return None

    async def bulk_update_users(
        self,
        updates: Dict[str, Dict[str, Any]]
    ) -> Tuple[List[User], List[str]]:
        """Update multiple users in bulk."""
        updated_users = []
        errors = []

        # Create tasks for concurrent updates
        tasks = []
        for user_id, update_data in updates.items():
            task = self._update_single_user(user_id,
update_data)
            tasks.append((user_id, task))

        # Execute updates concurrently
        for user_id, task in tasks:
            try:
                user = await task
                if user:
                    updated_users.append(user)
                else:
                    errors.append(f"User {user_id} not found")
            except Exception as e:
                errors.append(f"User {user_id}: {str(e)}")
        return updated_users, errors

    async def _update_single_user(self, user_id: str, updates:
Dict[str, Any]) -> Optional[User]:
        """Update a single user."""
        try:
            user = await User.get(user_id)

            for key, value in updates.items():
                if hasattr(user, key):
                    setattr(user, key, value)

            user.updated_at = datetime.utcnow()
            await user.save()
            return user

        except NotFoundError:
            return None

    async def search_with_concurrency(
        self,
        search_params: List[Dict[str, Any]]
    ) -> Dict[str, List[User]]:
        """Execute multiple searches concurrently."""
        tasks = {}

        for i, params in enumerate(search_params):
            task_name = f"search_{i}"
            tasks[task_name] = self._execute_search(params)

        results = await asyncio.gather(*tasks.values(),
return_exceptions=True)

        return {
            name: result if not isinstance(result, Exception)
else []
            for name, result in zip(tasks.keys(), results)
        }

    async def _execute_search(self, params: Dict[str, Any]) ->
List[User]:
        """Execute a single search with parameters."""
        query = User.find()

        # Apply filters based on parameters
        if 'is_active' in params:
            query = query.filter(User.is_active ==
params['is_active'])

        if 'min_age' in params:
            query = query.filter(User.age >=
params['min_age'])

        if 'max_age' in params:
            query = query.filter(User.age <=
params['max_age'])

        if 'interests' in params:
            interest_conditions = [User.interests << interest
for interest in params['interests']]
            if interest_conditions:
                combined = interest_conditions[0]
                for condition in interest_conditions[1:]:
                    combined = combined | condition
                query = query.filter(combined)

        if 'bio_search' in params:
            query = query.filter(User.bio %
params['bio_search'])

        # Apply limits
        limit = params.get('limit', 100)
        return await query.limit(limit).all()

    async def analyze_user_patterns_async(self) -> Dict[str,
Any]:
        """Analyze user patterns with async processing."""
        # Execute multiple analysis tasks concurrently
        tasks = {
            'age_analysis': self._analyze_age_patterns(),
            'interest_analysis':
self._analyze_interest_patterns(),
            'location_analysis':
self._analyze_location_patterns(),
            'activity_analysis':
self._analyze_activity_patterns()
        }

        results = await asyncio.gather(*tasks.values(),
return_exceptions=True)

        analysis = {}
        for name, result in zip(tasks.keys(), results):
            if isinstance(result, Exception):
                logger.error(f"Analysis {name} failed:
{result}")
                analysis[name] = {"error": str(result)}
            else:
                analysis[name] = result

        return analysis
    async def _analyze_age_patterns(self) -> Dict[str, Any]:
        """Analyze age distribution patterns."""
        users = await User.find().all()

        ages = [user.age for user in users]
        if not ages:
            return {"message": "No users found"}

        return {
            "total_users": len(ages),
            "average_age": sum(ages) / len(ages),
            "min_age": min(ages),
            "max_age": max(ages),
            "age_groups": self._group_by_age(ages)
        }

    async def _analyze_interest_patterns(self) -> Dict[str,
Any]:
        """Analyze interest distribution."""
        users = await User.find().all()

        interest_counts = {}
        total_interests = 0

        for user in users:
            for interest in user.interests:
                interest_counts[interest] =
interest_counts.get(interest, 0) + 1
                total_interests += 1

        top_interests = sorted(
            interest_counts.items(),
            key=lambda x: x[1],
            reverse=True
        )[:10]

        return {
            "total_interests": total_interests,
            "unique_interests": len(interest_counts),
            "top_interests": [{"interest": k, "count": v} for
k, v in top_interests],
            "average_interests_per_user": total_interests /
len(users) if users else 0
        }

    async def _analyze_location_patterns(self) -> Dict[str,
Any]:
        """Analyze location distribution."""
        users = await
User.find(User.address.is_not_null()).all()
```

```
        city_counts = {}
        state_counts = {}

        for user in users:
            if user.address:
                city = user.address.city
                state = user.address.state

                city_counts[city] = city_counts.get(city, 0) +
1
                state_counts[state] = state_counts.get(state,
0) + 1

        return {
            "users_with_location": len(users),
            "unique_cities": len(city_counts),
            "unique_states": len(state_counts),
            "top_cities": sorted(city_counts.items(),
key=lambda x: x[1], reverse=True)[:10],
            "top_states": sorted(state_counts.items(),
key=lambda x: x[1], reverse=True)[:10]
        }

    async def _analyze_activity_patterns(self) -> Dict[str,
Any]:
        """Analyze user activity patterns."""
        total_users = await User.find().count()
        active_users = await User.find(User.is_active ==
True).count()
        verified_users = await User.find(User.is_verified ==
True).count()

        # Users with recent login (last 30 days)
        recent_cutoff = datetime.utcnow() - timedelta(days=30)
        recent_active = await User.find(
            User.last_login >= recent_cutoff
        ).count()

        return {
            "total_users": total_users,
            "active_users": active_users,
            "inactive_users": total_users - active_users,
            "verified_users": verified_users,
            "recent_active_users": recent_active,
            "activity_rate": active_users / total_users if
total_users > 0 else 0,
            "verification_rate": verified_users / total_users
if total_users > 0 else 0
        }
    def _group_by_age(self, ages: List[int]) -> Dict[str, 
int]:
        """Group ages into ranges."""
        groups = {}
        for age in ages:
            group = f"{(age // 10) * 10}-{(age // 10) * 10 + 
9}"
            groups[group] = groups.get(group, 0) + 1
        return groups

async_user_service = AsyncUserService()
```

➢ **Sync Wrapper for Compatibility**

```
# services/sync_user_service.py
import asyncio
from typing import List, Optional, Dict, Any
from services.async_user_service import AsyncUserService
from models.user import User

class SyncUserService:
    """Synchronous wrapper for async user service."""

    def __init__(self):
        self.async_service = AsyncUserService()

    def _run_async(self, coro):
        """Run async coroutine in sync context."""
        try:
            loop = asyncio.get_event_loop()
        except RuntimeError:
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)

        return loop.run_until_complete(coro)

    def create_user(self, user_data: Dict[str, Any]) -> User:
        """Create a user synchronously."""
        async def _create():
            users, errors = await 
self.async_service.create_users_batch([user_data])
            if errors:
                raise ValueError(errors[0])
            return users[0]

        return self._run_async(_create())

    def find_by_id(self, user_id: str) -> Optional[User]:
        """Find user by ID synchronously."""
        async def _find():
            result = await 
self.async_service.find_users_parallel([user_id])
```

```
            return result.get(user_id)

        return self._run_async(_find())

    def create_users_batch(self, users_data: List[Dict[str,
Any]]) -> tuple:
        """Create multiple users synchronously."""
        return self._run_async(
            self.async_service.create_users_batch(users_data)
        )

    def search_users(self, search_params: Dict[str, Any]) ->
List[User]:
        """Search users synchronously."""
        async def _search():
            results = await
self.async_service.search_with_concurrency([search_params])
            return list(results.values())[0]

        return self._run_async(_search())

    def get_user_analytics(self) -> Dict[str, Any]:
        """Get user analytics synchronously."""
        return self._run_async(
            self.async_service.analyze_user_patterns_async()
        )

# For applications that need sync interface
sync_user_service = SyncUserService()
```