**Task Management Microservice Project Documentation**

**Project Overview**

This Task Management System implements a comprehensive microservice architecture to handle task creation, updating, querying, and visualization. The system leverages Redis for data storage and Flask for backend services, with services communicating through HTTP APIs. The application allows users to create tasks, update their status, query based on filters, and view dashboard statistics.

**System Architecture**

The system consists of four microservices:

1. **Dashboard Service**: Central hub for visualizing task statistics and system activity
2. **Task Creation Service**: Manages the creation of new tasks
3. **Task Update Service**: Handles modifications and deletions of existing tasks
4. **Task Query Service**: Provides task filtering, retrieval, and statistical information

All services interact with a shared Redis database for data persistence, and communicate with each other through REST API endpoints.

**Service Descriptions**

**Dashboard Service**

**Purpose**: Acts as the central visualization hub for the task management system.

**Functionality**:

* Displays current task statistics by status (pending, in-progress, completed)
* Shows task distribution by priority (low, medium, high)
* Tracks daily activity metrics (new tasks created, task updates)
* Provides notification endpoints for other services to report activity

**API Endpoints**:

* / - Main dashboard view
* /notify\_new\_task - Receives notifications of new tasks
* /notify\_task\_update - Receives notifications of task updates
* /notify\_task\_delete - Receives notifications of task deletions

**Task Creation Service**

**Purpose**: Manages the creation of new tasks in the system.

**Functionality**:

* Provides a user interface for task creation
* Generates unique IDs for each new task
* Stores tasks in Redis with appropriate metadata
* Notifies the Dashboard service when new tasks are created

**API Endpoints**:

* / - Task creation form
* /create - Processes task creation requests
* /success - Confirmation page after successful creation

**Task Update Service**

**Purpose**: Handles the modification and deletion of existing tasks.

**Functionality**:

* Provides an interface for editing task details
* Allows changing task status, priority, due date, and other properties
* Enables task deletion
* Notifies the Dashboard of all updates and deletions
* Tracks and reports update statistics

**API Endpoints**:

* / - Task edit interface (requires task\_id parameter)
* /save - Processes task updates and deletions
* /success - Confirmation page after successful update
* /delete\_success - Confirmation page after successful deletion
* /api/task\_updates - Provides task update statistics

**Task Query Service**

**Purpose**: Enables task retrieval, filtering, and statistical analysis.

**Functionality**:

* Lists all tasks with filtering options by status and priority
* Provides detailed views of individual tasks
* Offers bulk deletion functionality
* Generates task statistics for the Dashboard service

**API Endpoints**:

* / - Task listing and filtering interface
* /filter - Filtered task views
* /delete-all - Bulk task deletion
* /api/tasks - Returns all tasks as JSON
* /api/task/<task\_id> - Returns specific task details
* /api/tasks/stats - Provides task statistics

**Data Model**

Tasks are stored in Redis using the following structure:

* **Set**: tasks - Contains IDs of all tasks in the system
* **Hash**: task:{task\_id} - Contains all properties of a specific task:
  + id - Unique identifier (UUID)
  + title - Task title
  + description - Detailed task description
  + priority - Task priority (low, medium, high)
  + status - Current status (pending, in\_progress, completed)
  + due\_date - Task deadline
  + created\_at - Creation timestamp

Activity metrics use date-based keys:

* updates\_count:{YYYY-MM-DD} - Number of task updates on a specific date
* new\_tasks\_count:{YYYY-MM-DD} - Number of new tasks created on a specific date

**Communication Patterns**

1. **Service-to-Redis Communication**:
   * All services read from and write directly to Redis
   * Redis serves as the central source of truth
2. **Service-to-Service Communication**:
   * Task Creation → Dashboard: Notifies when new tasks are created
   * Task Update → Dashboard: Notifies when tasks are updated or deleted
   * Task Query → Dashboard: Provides the dashboard with statistical information
   * Dashboard → Task Query/Update: Retrieves statistics and activity data
3. **User-to-Service Communication**:
   * Users interact with specific services based on their intent (create, update, or query tasks)
   * The Dashboard provides a unified view of system activity and statistics

**Setup and Execution**

The application uses Docker Compose to orchestrate the services:

1. **Nginx**: Serves as the reverse proxy, routing requests to appropriate services
2. **Redis**: Provides in-memory data storage for all services
3. **Dashboard Service**: Runs on port 5000
4. **Task Creation Service**: Runs on port 5000
5. **Task Update Service**: Runs on port 5000
6. **Task Query Service**: Runs on port 5000

All services are containerized and can be started with a single command:

docker-compose up

**Technologies Used**

* **Flask**: Web framework for all backend services
* **Redis**: In-memory data storage
* **Docker & Docker Compose**: Containerization and orchestration
* **HTML/CSS**: Frontend interfaces
* **Python Requests**: API communication between services

This architecture demonstrates effective use of microservices, containerization, and stateless design principles while maintaining a cohesive user experience.