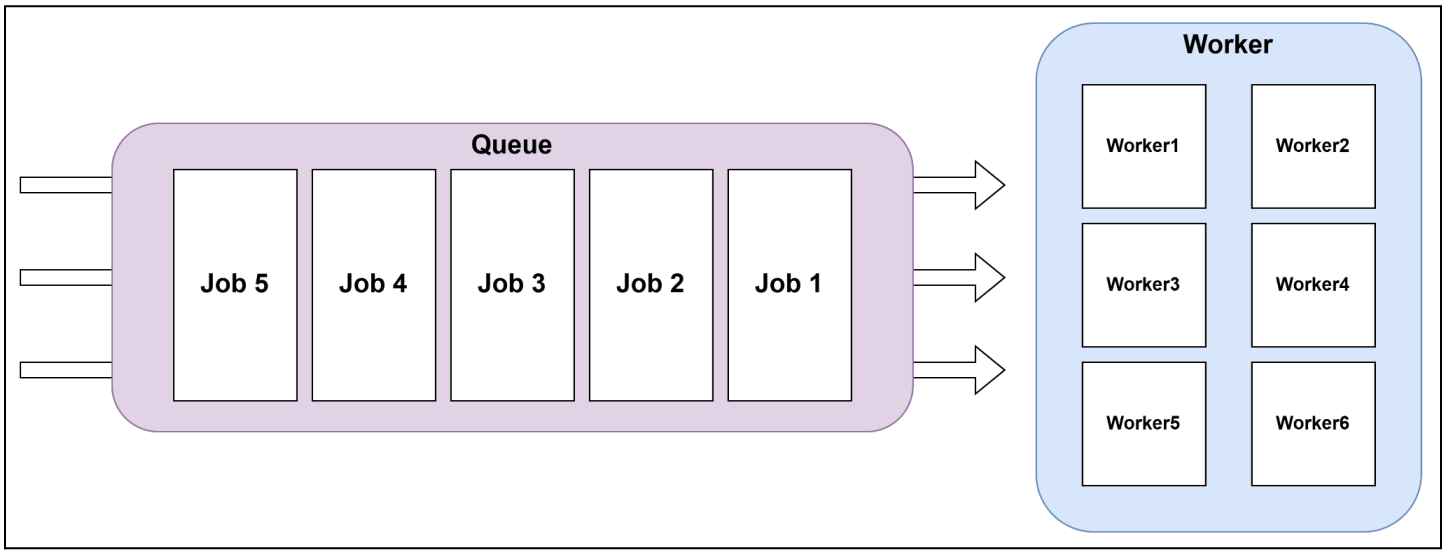
Simple Distributed Job Queue Simulation

By Alfandito Rais Akbar



Problem Author: arkanmis

Problem Challanges:
https://github.com/arkanmis/simple-distributed-job-queue
My Solution:
https://github.com/alfanditora/simple-distributed-job-queue

System Architecture

The system follows a modular architecture that separates responsibilities across components. Communication between core components is managed through interfaces and dependency injection.

Core Components:

1. **Entity**: Defines the core data structure representing a Job and its status.
2. **Interface**: Specifies contracts for data repositories and business services, allowing for flexible implementations.
3. **Repository/inmem**: An in-memory, concurrency-safe data repository used for job persistence and idempotency indexing.
4. **Service**: Contains business logic, including enqueuing jobs, managing idempotency keys, and retrieving job statuses.
5. **Worker**: The main component responsible for fetching, processing jobs, and updating their status. Implements polling and retry mechanisms.
6. **Main.go:** The application entry point, responsible for initializing all components, setting up the GraphQL API, and orchestrating the application lifecycle (including graceful shutdown).
7. **delivery/graphql**: Provides a GraphQL API interface for client interactions.

Component Implementation Details

`A. entity/job.go`

Defines the base structure for `Job` and `JobStatus`.

Job Fields:

- `ID (string)`: Unique job identifier.
- `Task (string)`: Name of the task to execute.
- `Token(string)`: Optional unique key to ensure the job is only processed once.
- `Status (string)`: Current job status (e.g., "pending", "running", "failed", "completed").
- `Attempts (int)`: Number of attempts made to process the job.

Job Status:

- Counts of jobs in each state: Pending, Running, Failed, Completed.

`B. interface/job_repository.go` and `interface/job_service.go`

Define contracts for data persistence and business logic layers.

JobRepository Methods:

- `FindAll(ctx)`, `FindByID(ctx, id)`, `Save(ctx, job)`, `FindByToken(ctx, token)`

JobService Methods:

- `Enqueue(ctx, taskName, idempotencyKey)`: Add a job to the queue.
- `GetAllJobs(ctx)`, `GetJobById(ctx, id)`, `SimultaneousCreateJob(ctx, taskNames, idempotencyKeys)`,
- `SimulateUnstableJob(ctx, taskName, idempotencyKey)`: Simulates jobs that fail and retry.
- `GetAllJobStatus(ctx)`: Aggregate status of all jobs.

C. `repository/inmem/job_repository.go`

In-memory implementation of `JobRepository`.
Structure:

- `mu sync.RWMutex`: Ensures thread-safe access to the in-memory store.
- `inMemDb map[string]*entity.Job`: Stores jobs by ID.
- `tokenIndex map[string]string`: Maps IdempotencyKeys to Job IDs for fast lookup.

Key Methods:

- `Save`: Persists or updates a job. Indexes the idempotency key.
- `FindByToken`: Retrieves job by its IdempotencyKey.

D. `service/job.go`

Contains the main business logic.
Enqueue:

- Auto-generates an IdempotencyKey if not provided (using taskName + UUID).
- Checks for an existing job using the key. Returns the existing job ID if found.
- Otherwise, creates a new job (status: pending, attempts: 0), and stores it.

SimulateUnstableJob: Creates a job designed to fail several times before succeeding.
GetAllJobStatus: Aggregates job statuses from the repository.

E. `worker/worker.go`

The background job processor.
WorkerOptions: Configurable settings (e.g., polling interval, max workers, retry delay, max attempts).
Start(ctx):

- Launches the main goroutine to poll for jobs periodically using a ticker.
- Uses `ctx` for shutdown signals.

pollAndProcessJobs(ctx):

- Fetches all eligible jobs (pending/failed with remaining attempts).
- Uses a worker pool (`workerPool chan struct{}`) to control concurrency.
- Processes each job in a separate goroutine.

processJob(ctx, job):

- Sets status to "running".
- Simulates task execution (e.g., via `time.Sleep`).

- Handles success or failure:
    - On failure: increments attempts, sets status back to "pending" or "failed" based on MaxAttempts.
    - On success: sets status to "completed".
- Saves the final status.

Stop(): Triggers a graceful shutdown, waiting for all running jobs to finish.

F. `main.go`

Main orchestrator of the application.

- Initialization: Sets up `JobRepository`, `JobService`, and `JobWorker`.
- GraphQL API: Uses Echo and `graph-gophers/graphql-go` for `/graphql` and `/graphiql` endpoints.
- Worker Start: Calls `jobWorker.Start(ctx)` to launch background processing.
- Graceful Shutdown:
    - Echo server runs in a separate goroutine.
    - Waits for OS signals (SIGINT, SIGTERM).
    - Calls `cancel()` to signal worker shutdown.
    - Gracefully shuts down Echo and calls `jobWorker.Stop()` to finish job processing before exit.

G. `delivery/graphql`

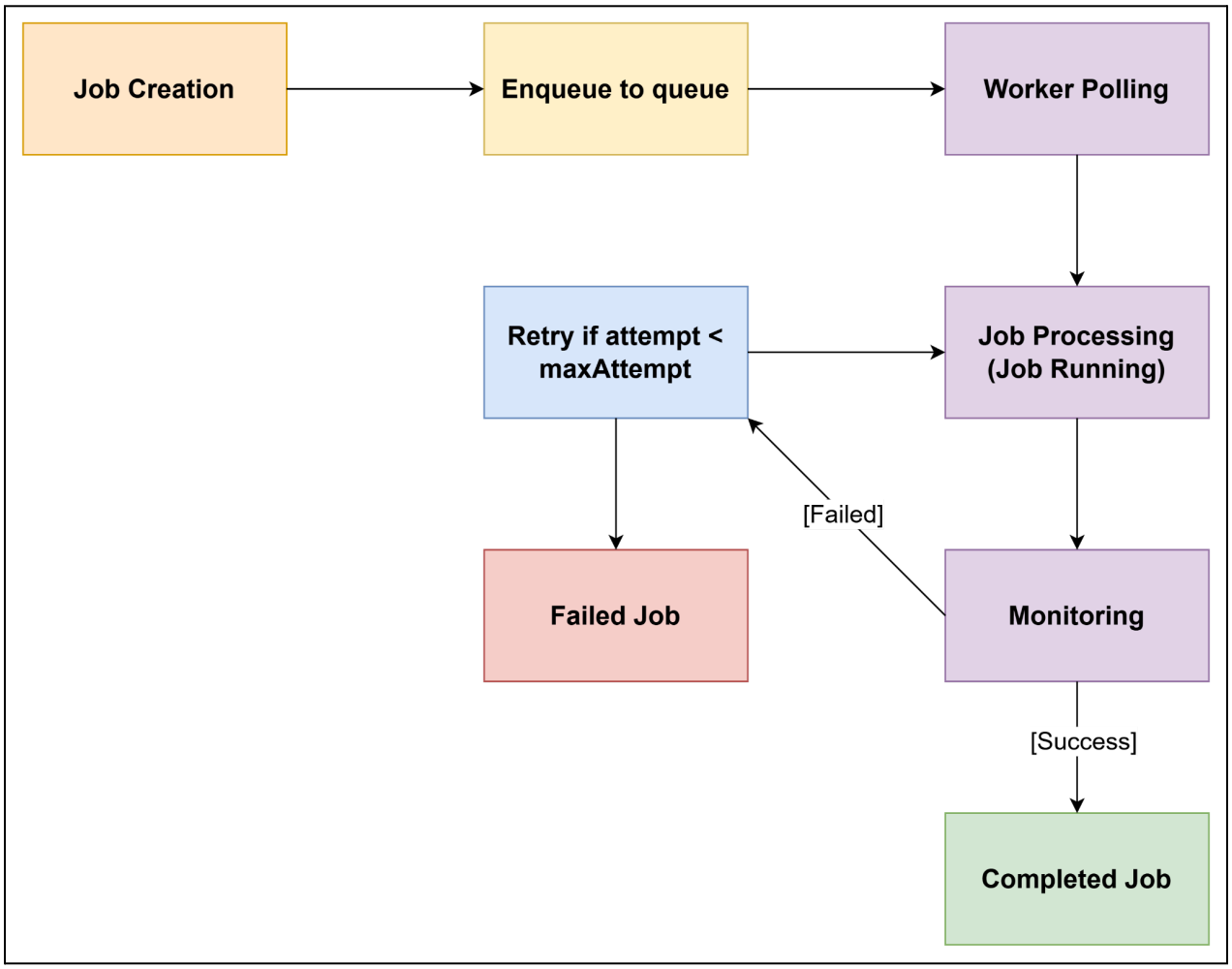
Exposes the GraphQL API for clients.

- **schema.graphqls: Defines GraphQL types, queries, and mutations (e.g., Enqueue).**
- **mutation/job_mutation.go: Resolver for mutations, calling the appropriate service methods.**
- **query/job_query.go: Resolver for queries, such as `GetAllJobs`, `GetJobById`, and `GetAllJobStatus`.**

Key Implemented Features

1. **Asynchronous Processing**: Jobs are enqueued and processed in the background, decoupled from API requests.
2. **Idempotency**: Prevents duplicate job processing using a unique key.
3. **Job Status Tracking**: Each job has a lifecycle status that updates as it progresses.
4. **Retry Mechanism**: Failed jobs are retried until they succeed or reach MaxAttempts, with delays between retries.
5. **Concurrency Control**: Limits parallel job processing using a worker pool.
6. **Graceful Shutdown**: Ensures all in-progress jobs complete before the system exits.
7. **In-Memory Persistence**: Job data is stored in memory for simplicity, but the repository can be swapped with a persistent backend like PostgreSQL or Redis.
8. **GraphQL API**: Modern and flexible API interface for client interaction.

Workflow Overview



1. **Job Creation**: Client sends a GraphQL mutation (`Enqueue` or `SimulateUnstableJob`).
2. **Service Layer**:
   - The resolver calls `jobService.Enqueue`.
   - If an existing job with the given IdempotencyKey exists, its ID is returned.
   - Otherwise, a new job is created with `pending` status and stored.
3. **Worker Polling**:
   - `JobWorker` periodically polls for `pending` or retryable `failed` jobs.
4. **Job Processing**:
   - Status is updated to `running`.
   - Task logic is executed (simulated).
   - If success → `completed`; if failed → `pending` or `failed` based on attempts.
   - Final status is saved.

5. **Monitoring**: Clients query GraphQL to track progress (`GetJobById`, `GetAllJobStatus`).
6. **Shutdown**:
    - On receiving a shutdown signal, the system stops accepting new requests and waits for all ongoing jobs to finish before exiting.

Testing Strategy

A. Unit Tests (`service/job_test.go`, `repository/imem/job_test.go`, `worker/job_test.go`):

- Test core service logic in isolation using `testify/mock` for mocking `JobRepository`.
- Scenarios include:
    - New job creation.
    - Idempotency behavior.
    - Error handling.
    - Status retrieval.

B. Stress/Integration Tests (`test/job_worker_stress_test.go`):

- Test end-to-end system behavior under load.
- Enqueue a large number of jobs (e.g., 100 including unstable jobs).
- Monitor and verify all jobs complete (either `completed` or `failed`) without crashes or deadlocks.
- Measure processing time to assess performance.