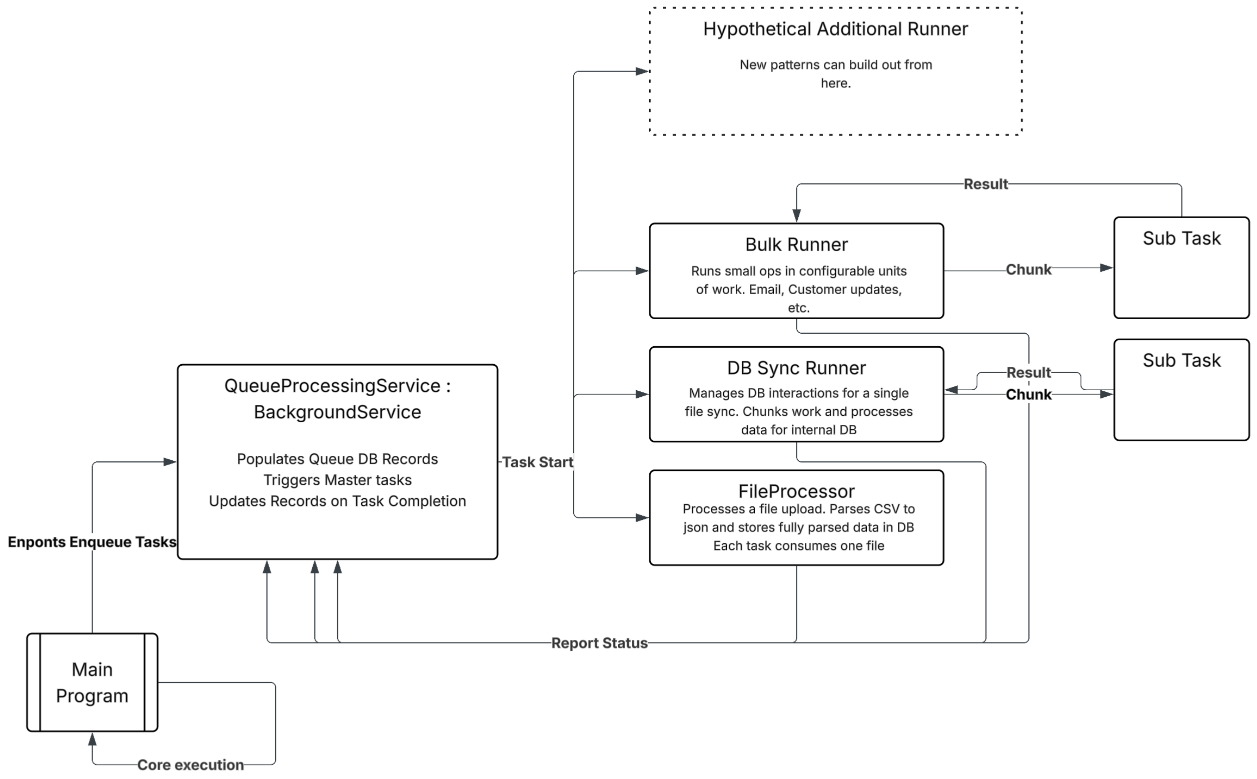
This is an overview of a proposed concurrent background worker system in .Net core intended to manage the workflow of several different background tasks for a large system of users of a car wash application.

Background Worker System Overview


Hypothetical Additional Runner
New patterns can build out from here.
Result
Bulk Runner
Runs small ops in configurable units of work. Email, Customer updates, etc.
Chunk
Sub Task
DB Sync Runner
Manages DB interactions for a single file sync. Chunks work and processes data for internal DB
Result
Chunk
Sub Task
QueueProcessingService : BackgroundService
Populates Queue DB Records
Triggers Master tasks
Updates Records on Task Completion
Task Start
FileProcessor
Processes a file upload. Parses CSV to json and stores fully parsed data in DB
Each task consumes one file
Enponts Enqueue Tasks
Report Status
Main Program
Core execution

Component Overview

- **Controller**
  - Standard API endpoints. One per TaskType
    - Each endpoint will create an object of type `QueueTask<T>` and enqueue it where T is the `TaskType` sub-type. This sub type will contain the details required for the specific task to execute (a list of customer records to edit, email details, files to process, etc), and is stored in the `worker_queue` DB table as a JSONB.
  - Additional Endpoints
    - FetchTasksFromDB:
      - tell the queue to retrieve tasks from the DB and process any that are new/need retries/etc.
    - Retry Failed Jobs

        - Triggers a re-run of all failed jobs in the queue.
        - additional parameters for time ranges and specific task ID's
    - Rerun job
        - Re-fire a job that has already completed. This could take one of a couple of forms, but the easiest approach is probably to clone the task.
- **Queue**
    - The queue itself runs as an `UnboundedChannel`. Standard Enqueue and Dequeue Methods plus a method for checking if something exists in the queue currently.
    - The queue is a singleton, and is injected into the queue service.
    - When objects are enqueued also stores a record of that task in the DB. We Generate a custom ID for that task. Process ID is included in the message stack at instantiation of a task and when sub tasks are generated.
    - At a set interval, fetch from DB any unprocessed tasks and put them in the queue.
- **QueueProcessor**
    - This is the main processor for the worker queue.
    - Implements BackgroundService
    - the main loop listens to the queue and dequeues tasks from it when they exist. Based on the task type, it passes the object off to the individual service callers
    - Each caller invokes Task.Run and generates a new thread that handles the queue task.
- **Individual Services**
    - This is a rundown of the general approach I would take to implement each of these services at a very high level, hitting a few possible pain points and a general approach I would take.
    - Individual Task Types:
        - **Email**
            - Type: Bulk simple processing
            - Trigger: Admin Action, CRON
            - Considerations:
                - Needs retries because email services are weird and unreliable
                - Too many at once gets flagged as spam
                - Bulk emailing can be slow while waiting for status from SMTP if not handled well
            - **Work Strategy**:
                - Chunk work into a size that doesn't cause the task to run too long. Add support for retries and good error reporting. Throughput management may rely on 3rd party SMTP service features to some extent.

- **Customer Credits:**
    - Type: Bulk simple processing
    - Trigger: Admin Action
    - Considerations:
        - Easier than email. Internal DB operations.
        - Probably the lightest weight of any of the TaskTypes;. Might be good for a junior/mid-level engineer to get their feet wet.
    - **Work Strategy**:
        - Same as email more or less. Chunk it up and process it. Extra consideration for additional DB interactions. Maybe a master runner that processes the whole set
- **Financial DB Sync**
    - Type: Continuous task
    - Trigger: CRON
    - Considerations: Long running, large data set processing. Incremental?
    - Assumptions:
        - This is not a live-data retrieval operation.
    - Strategy:
        - If this is a daily/nightly sync, we should consider scheduled DB jobs to do the ETL work. ETL would allow us to get a stable snapshot for processing records in a way we know wont change under us while we process.
        - work is chunked into a processing set, cursor (or other record tracking) is stored in the output of the job. This task
- **File upload**
    - Type: Parsing & Data Transformation, I/O
    - Trigger: Admin Action
    - Considerations:
        - Employee data will never be that large, each file can probably be processed in memory in totality
        - Implementation needs to be aware of locking up DB resources too much when adding/editing records. We don't want to run into locks
        - This may be a good candidate for a nightly job situation if we don't need the data immediately.
    - Strategy:

        - Pull the file from Azure Blob Storage. Parse the CSV into a data structure. store that object in the database. pass the task back with results.

Workflow

Adding New Runners

The process of adding an additional runner would be relatively simple:

1. Add new TaskType Data model
2. Create a new endpoint to enqueue new task type
3. Create a service that manages the logic of the operation
4. update the QueueProcessor to handle the new type and call the new type handler.