Site Management System

This is a system designed to manage items and associated images for multiple websites. Each item can have various types of images: static images, preview images, and production images. The system provides functionality to delete multiple items and their associated images while considering certain constraints and operational considerations.

Next, find a brief description of some system design decisions. This does not pretend to be an exhaustive nor a complete document – these are notes of the high level design.

The data model

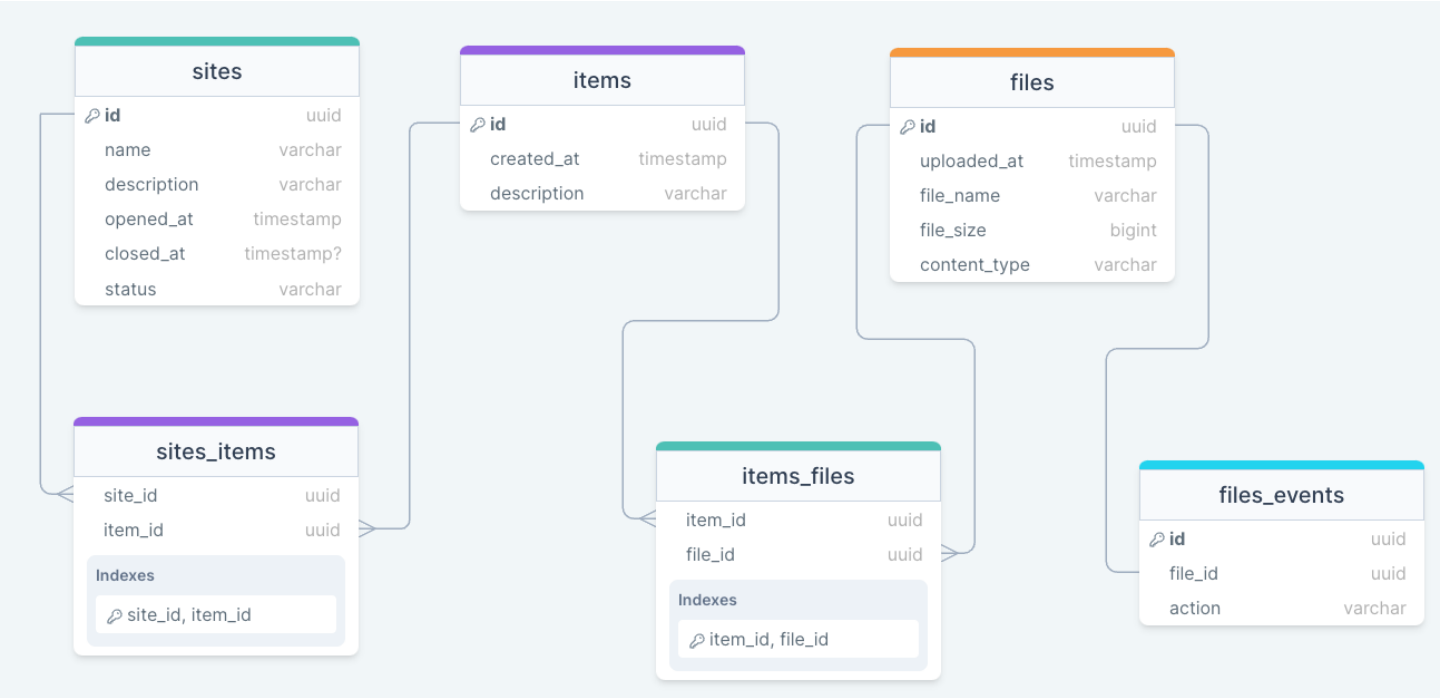

Figure 1 - ER Diagram

```
CREATE TABLE "sites"(
    "id" UUID PRIMARY KEY NOT NULL,
    "name" VARCHAR(255) NOT NULL,
    "description" VARCHAR(255) NOT NULL,
    "opened_at" TIMESTAMP NOT NULL,
    "closed_at" TIMESTAMP,
    "status" VARCHAR(255) NOT NULL
);

CREATE TABLE "items"(
    "id" UUID PRIMARY KEY NOT NULL,
    "created_at" TIMESTAMP NOT NULL,
    "description" VARCHAR(255) NOT NULL
);

CREATE TABLE "sites_items"(
    "site_id" UUID NOT NULL,
    "item_id" UUID NOT NULL,
    PRIMARY KEY("site_id", "item_id"),
    FOREIGN KEY("site_id") REFERENCES "sites"("id"),
    FOREIGN KEY("item_id") REFERENCES "items"("id")
);
```

```
CREATE TABLE "files"(
    "id" UUID PRIMARY KEY NOT NULL,
    "uploaded_at" TIMESTAMP NOT NULL,
    "file_name" VARCHAR(255) NOT NULL,
    "file_size" BIGINT NOT NULL,
    "content_type" VARCHAR(255) NOT NULL
);

CREATE TABLE "items_files"(
    "item_id" UUID NOT NULL,
    "file_id" UUID NOT NULL,
    PRIMARY KEY("item_id", "file_id"),
    FOREIGN KEY("item_id") REFERENCES "items"("id"),
    FOREIGN KEY("file_id") REFERENCES "files"("id")
);

CREATE TABLE "files_events"(
    "id" UUID PRIMARY KEY NOT NULL,
    "file_id" UUID NOT NULL,
    "action" VARCHAR(255) NOT NULL,
    FOREIGN KEY("file_id") REFERENCES "files"("id")
);
```

System Design Notes

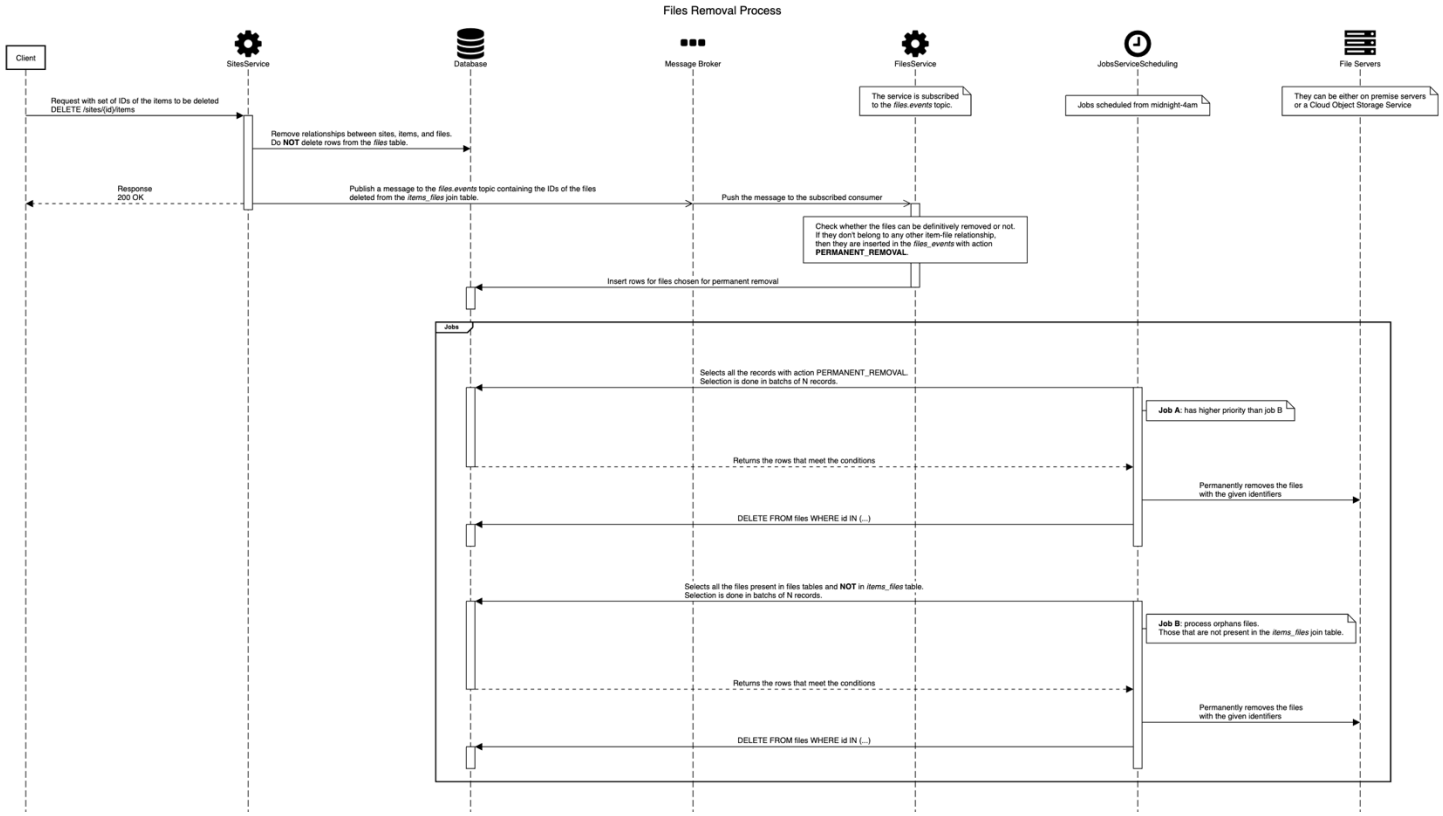

Figure 2 - Sequence Diagram for the Files Removal Process

You can zoom in to better see the diagram, in any case, you can also find it full size here: https://t.ly/site-management-system-sequence-diagram

If after reading the sequence diagram, you still have questions, find some additionals details here:

- The main service serving the requests is a RESTful API over HTTP, let's call it: `SitesService`. This service provides an endpoint (`DELETE /sites/{id}/items`) that receives a set of the IDs of the items to be deleted. The service:
  - Performs a DB deletion of all the items and the associations that they could have in the *items_files* table.
  - After that, it asynchronously publishes an event with a message that contains the IDs of all the deleted files, those that were removed from the join table. The message is published to the *files.events* topic.
  - Then, it returns with a 200 response if everything was OK.
- The `FilesService`, is subscribed to the *files.events* topic and each time a new message arrives, it process it taking into consideration:
  - Static images: as they are not shared, they are inserted in the *files_events* table with action **PERMANENT_REMOVAL**.
  - Preview images: if no occurrences in the *items_files* table, it means no item have them associated, so it's safe to delete them. They are inserted in the *files_events* table with action **PERMANENT_REMOVAL**.
  - Production images: these are skipped and are candidates for the *orphan_images_removal_job* (Job B, in the sequence diagram).

This is a distributed system designed for high performance, fault tolerance, and hight availability. We should avoid single points of failures, that's why it's important to have:

1. Cache mechanisms in place (most requested sites, items, images, etc.) and, probably, a CDN would be required to speed up responses.
2. The services (`SitesService`, `FilesService`, etc.) in a cluster behind a load balancer that distributes the traffic load.
3. Retries for partial/transient network calls failures, this should be implemented along with:
    a. Timeouts as strict as possible for connections and requests.
    b. Backoff to avoid overloading servers – exponential backoff is recommended.
    c. Idempotent APIs, so a request can be safely retried.
4. Read database replicas that can be used by the jobs and other services.

The nature of the system allows to have eventual consistency instead of a strong consistency. In place of a SQL database, we could use a NoSQL database that handles partition-tolerance, so we could have a more robust, performant and resilient system – DynamoDB could be an option.

This is already a complex system, and we should keep it simply, right? Well, this is a theoretical exercise that can be perfectly implemented, however, yes, it will require more time and it's probably an overkill for lots of scenarios.

In the code, I have implemented each service as a class to keep it simple. This can be a monolithic application without a doubt.

Further improvements

- Data sharding – an anylisis for this is required. How will be the limits for each batch of the jobs defined? Will there be improvements if the data is sharded by, let's say, sites location? What other criterias for sharding could we use?
- Observability – logs, metrics, traces, and alerting. How fast can we react to failures? How fast can we diagnose those failures? Can I trace a request end to end to better understand, for example, performance bottlenecks? Can I use the metrics to trigger alerts?