The Photo Union

for the Advancement of Software and Photo Art

Photocol (backend)
Contributors: Jonathan Lam, Richard Lee, Tiffany Yu, Victor Zhang
GitHub: https://github.com/photocol/photocol-server

Macro-level Overview of the Project
The main server is stored in photocol-server.
DB setup/maintenance operations are stored in photocol-DB_SETUP.
The CLI is stored in photocol-cli.
Specifics about endpoints, error codes, and more are available in the server wiki.

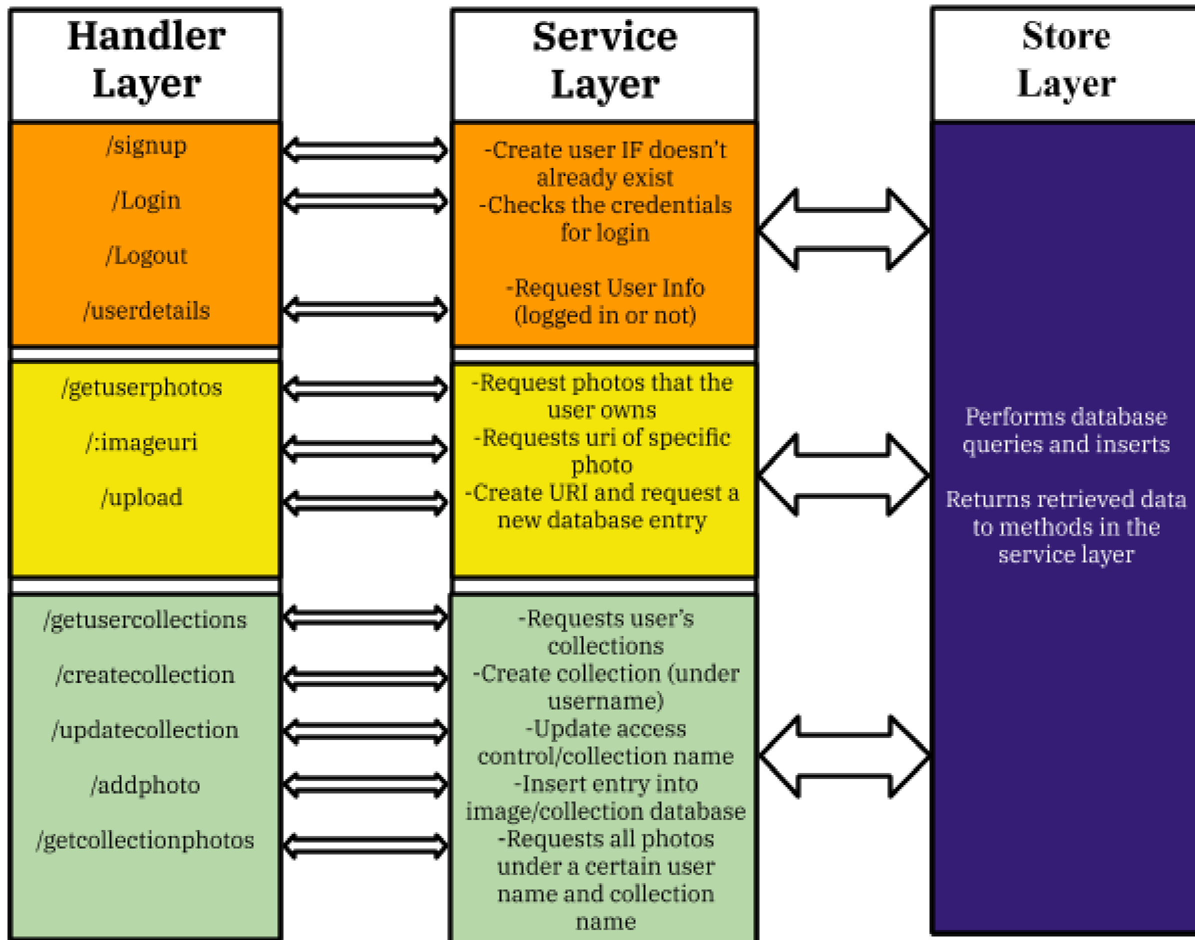
Dependencies: Spark Framework, Gson, SLF4J, Apache Commons 3, Amazon AWS SDK V2, MariaDB JDBC driver. Maven is used to manage dependencies.

User-oriented Overview of the System
The Photo Union is a photo management and collection system, where users can create accounts, upload photos and make collections to share with others.

At the current stage of the project, users can interact with the server via a custom CLI (photocol-cli), or via ordinary HTTP requests. Current features implemented include basic user creation and login session management, ability to create photo collections, ability to upload photos to account, ability to add photos to collections, and ability to get photos from the server, all with basic permissions checking.

Users have roles in collections: Owner, Editor, and Viewer as a form of access control. Images cannot be shared by themselves; rather, access permissions are set on entire collections. Image names (once uploaded to S3) are randomly-generated and unique site-wise; collections are accessible through a semantic URL namespaced by username (e.g., /someuser/theircollectionname, where someuser2 could also have a collection called theircollectionname). (Usernames are unique.)

Black-Box Diagram



High Level Overview of Codebase
Our system has three layers, the handler layer, the service layer, and the store layer. This type of abstraction allows for different team members to work on different aspects of the design without getting too hung up with all the aspects of the system.

The handler layer (package photocol.layer.handler) is responsible for responding to the HTTP requests from the end-user, deserializing and doing (currently minimal) validation on the user, and passing the deserialized objects down to the service layer. (Only the /logout endpoint doesn't pass down data to the next layer, since it only invalidates the http session).

The service layer (package photocol.layer.service) is responsible for the application logic. Some of these endpoints are a simple passthrough (e.g.,

PhotoService::getUserPhotos) to the store layer, but many of these perform multiple store-level logic (i.e., checking and validation).

The store layer (package photocol.layer.store) manages the database. This layer provides methods to interface with retrieve and insert data into the databases. See comments below about why we decided to use a database.

In addition, we have some utility classes (package photocol.util), which contains a MariaDB and AWS S3 connection manager.

This layered structure also enables us to have 'hot swappable' layers - any changes to the one layer are opaque to another layer. This means that upgrades to each layer are possible without breaking the entire system. The trade-off is that we have to be careful when making changes to our method, that is keep the return types and return values in sync.

Design Decisions (a slightly-lower-level overview)
While databases were out of scope, we decided to implement a MySQL (actually: MariaDB) DB scheme, because we felt that that would be the most logical way to implement all of the access control schemes, especially with joins and built in concurrency management. In particular, this means that in addition to basic tables storing user, photo, and collection data, in order to implement the many-to-one relationship between images to users, we use foreign keys in the photo table (currently not implemented as foreign keys, but that will be changed later). Similarly, for the many-to-many implementation between users and collections (i.e., access control permissions), and the many-to-many implementation between collections and images, we create two additional junction tables just storing lists of relations (pairs of foreign keys), which will be used in joins to check user permissions to images or collections.

Also, we decided to implement S3 capability for uploading/downloading, since this was an integral part of our system and provided a scalable interface for large volumes of uploads and downloads (as opposed to uploading directly onto our server, for which we have not received the specifications of and may not be suited to large network traffic). S3 has the nice-to-have's of huge scalability, high reliability, version control (e.g., ETags for caching/change detection), and concurrency support built-in.

Data and Entity Schemas
Common Java entity classes and their public members are shown. Most of these Java types are only meant to manage the fields of an incoming or outgoing request, and meant to be (more or less) directly serialized/deserialized to/from JSON.

class photocol.definitions.response.StatusResponse<T>: a wrapper for returning status codes with optional data.

enum Status: making application-specific response codes semantic

int status: hold the Status enum response code
T payload: hold the (optional) data payload of the request

class photocol.definitions.User: hold credentials on signup/login requests, or when retrieving user data

String email
String username
String passwordHash (currently-unhashed)

class photocol.definitions.PhotoCollection: hold information about a collection when performing some operation about it

boolean isPublic
String name
List<ACLEntry> aclList: list of access control entries, only for endpoints that request to change/view the access control list to the collection
String uri: this is automatically generated from name to allow collections to be accessible at /collections/:collectionuri

class photocol.definitions.Photo: hold information about an image when updating or retrieving image information

String uri
String description
String uploadDate

The schema for the database tables is shown below. These definitions can be found in the photocol-DB_setup repo.

table photocol.user:

uid int not null auto_increment
email varchar(255) not null unique
username varchar(255) not null unique
password varchar(255) not null
primary key(uid)

table photocol.collection

cid int not null auto_increment
pub tinyint(1) not null
name varchar(255) not null

```
    uri varchar(255) not null
    primary key(cid)

table photocol.photo
    pid int not null auto_increment
    uri varchar(255)
    upload_date date
    description varchar(255)
    uid int not null
    primary key(pid)

table photocol.acl (Access Control Lists)
    cid int not null
    uid int not null
    role int not null
    primary key(cid, uid)

table photocol.icj (Image-Collection Junction table)
    pid int not null
    cid int not null
    primary key(pid, cid)
```

Team Dynamic
The project is definitely moving along rather well - a lot of functionality is coming together. We were/are having some version control/git syncing issues at points - it seems that IntelliJ doesn't automagically update files when we git pull (we have to click Refresh from Disk often). Also, since we are running 4 different OSes across the 4 of us, we do have some package version differences. MariaDB on one of our machines has a 767 max characters per row, which differs from the other three machines, and over the duration of this stage of the project, we were unable to resolve it. We may just have to recompile and reinstall a newer MariaDB.

In terms of team dynamic it seems we are *very* busy people with rather conflicting schedules so meeting up for during the week is a rather difficult task to accomplish. This means that we did have points in our project where two people wrote code for the same functionality because of improper communication. In the future we do need a better way of managing who does what and when in order to not waste anybody's time.