

*Hand in your code by copying your entire project directory (the `photoalbum` directory within your workspace directory) into your handin folder `/classes/cs343/handin/username` (where `username` is replaced by your username). **Rename the handed-in folder to `photoalbum-12a` for your first handin and `photoalbum-12b` for your second handin, if any** — don't delete or overwrite an earlier handin with a later one.*

These problems will be collected on Monday 10/26 and Monday 11/2.

Homeworks #12–13 involve the development of an API for a simple photo management system. Each user can maintain an album of photos. Photos may be tagged with keywords (a given photo may have multiple keywords) and, optionally, have a caption. Users may view all of the photos in their album or just those with a certain keyword, view a single photo and its keywords, add photos to their album, delete photos from their album, and set or change a photo's caption. Optionally, users may also list all of the keywords used in their album, add keywords to a photo, and replace a photo's entire set of keywords at once.

Users are identified by a username and password — eventually they will have to log in before viewing or modifying an album, but for the basic photo album, there is no login. Instead, each request simply names the user whose album it works with, so anyone can view or modify anyone's album. Creating accounts, logging in, and logging out will be added in homework #13.

Database

Use the database `username_photoalbum` in MySQL. The connection URL is `jdbc:mysql://bridgeman-vm.hws.edu:3306/username_photoalbum`. (In both cases `username` should be replaced by your username.) Note that you will need to connect to MySQL using the account you were given, not `guest`.

The database has been set up with some sample data and two users: `arthur` and `ford`. `arthur` has an album with 17 photos; a few (such as photo 1) have captions. `ford`'s album is empty. You can connect using MySQL Workbench to browse (or modify) the data and run queries directly. The image URLs are valid URLs — visit <http://bridgeman-vm.hws.edu/cs343f26/pics> to browse the collection. You can also use URLs of your own.

The database schema:

```
USER(username,password)
ALBUM(id,username,url,caption)
KEYWORD(id,keyword)
```

`USER` stores information about users and their passwords, `ALBUM` links photos to users, and `KEYWORD` links keywords to photos. `id` is a numerical ID (auto increment) used to uniquely identify entries in the `ALBUM` table. There is also a `SESSION` table, but that won't be used until homework #13.

Foreign keys:

- `ALBUM.username` → `USER.username`
- `KEYWORD.id` → `ALBUM.id`

Other constraints:

- All foreign key constraints are `ON DELETE CASCADE`.
- `(ALBUM.username, ALBUM.url)` is `UNIQUE`.
- All columns except `ALBUM.caption` are `NOT NULL`.

Provided Code and Setup

You have been provided with a starter framework containing a skeleton main program, common boilerplate code for database connections and error handling, and the package structure for organizing all of the classes. Except for stripping the application-specific code out of `Main`, these are the same classes (and the same organization) as in the library demo.

1 Setup

This assignment (and your project) relies on several external libraries — Javalin for the web server, Jackson for JSON, the MySQL Connector/J driver, and bcrypt for passwords — and those in turn depend on dozens of other jar files. As a result, we'll be using Maven, a build tool for Java projects that manages all these dependencies for you.

- Create a Maven project in Eclipse: choose `File` → `New Maven Project`. Check `Create a simple project` (skip archetype selection) (leave `Use default Workspace location` checked) and click `Next`. Fill in:
 - Group Id: `cs343`
 - Artifact Id: `photoalbum` (this becomes the project's name)

Leave the rest as it is and click `Finish`. The new project has the standard Maven layout, including a `src/main/java` folder for your code and a placeholder `pom.xml`. (`pom.xml` contains all of Maven's configuration.)

- Replace `pom.xml`: import `/classes/cs343/photoalbum/pom.xml` into the top level of your project, replacing the Eclipse-generated version. Open it and verify there are more than a few lines.

- Update the project: right-click on the project in the Package Explorer → Maven → Update Project..., verify that (only) the `photoalbum` project is checked under Available Maven Codebases, and click OK. This will update the JRE System Library to use Java 21, as configured in the provided `pom.xml`.
- Import the starter code: import the directory `/classes/cs343/photoalbum/photoalbum` and its contents into `src/main/java` within your project. To get things in the right place, right-click on `src/main/java` under your project in the Package Explorer and choose Import Navigate to `/classes/cs343/photoalbum` for the From directory, click `photoalbum` in the left side box to select it and everything it contains, then click `pom.xml` in the right side box to *unselect* it. (Click the little triangle next to `photoalbum` to reveal its contents — you should see another `photoalbum` directory which contains directories `dao`, `db`, etc directly under `photoalbum`.) Verify that Into folder says `photoalbum/src/main/java` and click Finish. Check in the Package Explorer — under `src/main/java` you should see `photoalbum`, `photoalbum.dao`, `photoalbum.db`, etc with Java files contained under each.

There shouldn't be any compiler errors.

- Set up the environment variables for connecting to the database: first run Main (right-click on Main, in the `photoalbum` package under `src/main/java`, → Run As... → Java Application. Javalin should start up. Stop the application (click the red stop button), then open the run configurations: right-click on Main → Run As... → Run Configurations... Click on the Environment tab and use Add... to add the following:
 - variable: `DB_USER`, value: your MySQL username
 - variable: `DB_PASSWORD`, value: your MySQL password
 - variable: `DB_URL`,
value: `jdbc:mysql://bridgeman-vm.hws.edu:3306/username_photoalbum`
where *username* is replaced by your MySQL username

Then click Apply and Close.

You'll test the connection a bit later.

2 Look Around

Look through the provided code:

- `db/DB.java` opens database connections, using the `DB_URL`, `DB_USER`, and `DB_PASSWORD` environment variables set in your run configuration.
- `errors/` holds the application's exceptions. Each names a kind of failure (`NotFoundException`, `ValidationException`, and so on), not an HTTP status code.

- `dao/SqlErrors.java` turns an `SQLException` into one of the application's exceptions.
- `routes/ErrorHandler.java` turns the application's exceptions into HTTP status codes and error JSON such as `{"status": 404, "message": "No photo with id 999999"}`.
- `routes/Params.java` parses the `{id}` path parameter, producing a 400 if it isn't a number.
- `Main.java` has `TODO` comments marking where each layer gets created and where routes get registered.

`photoalbum.dto` also contains two classes (`LoginRequest` and `TokenResponse`) that support authentication and authorization — ignore those for now, you'll use them in homework #13.

Endpoints

Listed below are six required endpoints and three optional endpoints. You are encouraged to implement the optional endpoints for the practice, but they are not required for completion of the homework.

Conventions that apply to every endpoint:

- Request and response bodies are JSON. Every error response has the same format: `{"status": 404, "message": "..."}.`
- `{id}` must be numeric (otherwise the result is a 400 response). A photo that doesn't exist and a photo that belongs to another user both produce a 404 response.
- The “current user” is the user whose photos an endpoint works with. Until authentication is added, the client names the current user in a header, `X-User: arthur`. A missing `X-User` header or one that names a user who doesn't exist produces a 401 response.
- Any unexpected failure produces a 500 response. This is not listed separately below.

Required Endpoints

Endpoint	Description	Request	Success response	Errors
GET /photos GET /photos?keyword=...	List the current user's photos, ordered by id. With keyword , list only the photos that have that keyword.	Query: keyword (optional)	200 [{"id": 12, "url": "http://...", "caption": "Ruapehu"}, {"id": 13, "url": "http://...", "caption": null}] ([] if the album is empty or no photo has the keyword)	401
POST /photos	Add a photo to the current user's album.	Body: {"url": "http://...", "caption": "Ruapehu"} (caption optional)	201, header Location: /photos/23 {"id": 23, "url": "http://...", "caption": "Ruapehu"}	400 malformed JSON; url missing, blank, over 500 characters, or not http://https://; caption over 200 characters 401 409 photo already in the album
GET /photos/{id}	Get one of the current user's photos.	Path: id	200 {"id": 12, "url": "http://...", "caption": "Ruapehu"}	400 non-numeric id 401 404

Endpoint	Description	Request	Success response	Errors
PUT /photos/{id}/caption	Set or replace a photo's caption; null, "", or a missing caption field clears it.	Path: id Body: { "caption": "Mt Ruapehu at dawn" }	200 { "id": 12, "url": "http://...", "caption": "Mt Ruapehu at dawn" }	400 non-numeric id; malformed JSON; caption over 200 characters 401 404
DELETE /photos/{id}	Delete a photo and its keywords.	Path: id	204 (no body)	400 non-numeric id 401 404
GET /photos/{id}/keywords	List one photo's keywords, alphabetically.	Path: id	200 ["Ruapehu", "volcano"] ([] if none)	400 non-numeric id 401 404

Optional Endpoints

Endpoint	Description	Request	Success response	Errors
POST /photos/{id}/keywords	Add a keyword to a photo.	Path: id Body: { "keyword": "Ruapehu" }	201 { "id": 12, "keyword": "Ruapehu" }	400 non-numeric id; malformed JSON; keyword blank or over 50 characters 401 404 409 photo already has that keyword
GET /keywords	List the distinct keywords on the current user's photos, alphabetically.	None	200 ["New Zealand", "Ruapehu", "volcano"] ([] if none)	401
PUT /photos/{id}/keywords	Replace a photo's entire keyword set.	Path: id Body: { "keywords": ["Ruapehu", "volcano"] } (duplicates ignored; [] removes all)	200 ["Ruapehu", "volcano"] (alphabetical)	400 non-numeric id; malformed JSON; keywords field missing; a keyword blank or over 50 characters 401 404

API Implementation

Instead of building the photo album API one layer at a time — every DAO method, then every service method, then every route — build it in *vertical slices*. A *slice* is one endpoint implemented through every layer — domain object, DAO, service, route, and `Main` — and then run and tested before you start the next one. This means that any configuration issues or problems in how the layers fit together are found and fixed early, before you've written a lot of code based on something you then have to change.

The first slice is the biggest as it is where you set up the pattern and write a lot of the supporting code; later slices repeat the same steps with only incremental additions. For this homework you've been provided with most of the files for the first slice so you don't have to write very much, however you should look through that code to make sure you understand it.

GET /photos/{id}

GET /photos/{id} retrieves information about one of the current user's photos. For example, the request

```
GET /photos/1
X-User: arthur
```

will produce a 200 response whose body contains the information about the photo:

```
{
  "id": 1,
  "url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_0081.JPG",
  "caption": "Mount Ngauruhoe, Tongariro National Park"
}
```

Implementing this first slice involves creating five new classes and editing `Main`. Work from the bottom up — the data first, then the business logic and rules, then HTTP — which is the same order `Main` creates the layers in:

File	What it does
domain/Photo.java	one photo, as a Java object
dao/PhotoDao.java	the SQL that finds a photo
service/PhotoService.java	business logic and rules (e.g. no such photo is an error)
routes/CurrentUser.java	determines whose album a request is for
routes/PhotoRoutes.java	the HTTP side of the endpoint
Main.java	creates the layers and connects them

3 The Domain Object — Photo

A `Photo` holds one row of the `ALBUM` table.

- Create `Photo.java` as shown in listing 1 (in the appendix). It goes in package `photoalbum.domain`.

Things to notice about the code:

- `Photo` is defined as a record, which gives you the constructor, the accessors (`photo.id()`, `photo.url()`, and `photo.caption()`), `equals`, and `toString` for free. Jackson also turns a record into JSON directly, with one field per component.
- `id` is an `Integer` rather than an `int` because `id` will be `null` for a photo that hasn't been inserted yet.
- The owner (`ALBUM.username`) isn't included because the DAO only ever returns photos belonging to the current user so every `Photo` has a known owner.

4 The DAO — PhotoDao

The DAO is the only layer that contains SQL.

- Create `PhotoDao.java` as shown in listing 2 (in the appendix). It goes in the package `photoalbum.dao`.

Things to notice about the code:

- The values go into the SQL through `?` placeholders and `setInt`/`setString`, never by building the SQL string directly from the values. That's key and is what protects the query from SQL injection.
- `AND username = ?` is the ownership check. If the photo belongs to someone else, the query finds no row — exactly as if the photo didn't exist.
- The try-with-resources statements close the result set, the statement, and the connection in every case, including when an exception is thrown.
- “No such row” isn't an error at this layer. The method returns an empty `Optional` and the service decides what that means.
- A real database failure becomes an application exception through `SqlErrors.translate` — no other layer ever sees an `SQLException`.
- `toPhoto` is a separate method because later slices (such as `GET /photos`) turn rows into photos too.

To verify that the DAO is working — that you can connect to the database and that the SQL is correct — add a `main` method to `PhotoDao` that prints

```
new PhotoDao().findById(1, "arthur")
```

and run it. Note that you'll need to set up a run configuration with the `DB_USER`, `DB_PASSWORD`, and `DB_URL` environment variables! You should see the following result:

```
Optional[Photo[id=1, url=http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_0081.JPG,
  caption=Mount Ngauruhoe, Tongariro National Park]]
```

You can leave `main` there for future testing purposes (it's a handy way to check the database connection) or delete it when you are done.

5 The Service — PhotoService

The service holds the application's rules. For this endpoint there's only one: asking for a photo that isn't in your album is an error.

- Create `PhotoService.java` as shown in listing 3 (in the appendix). It goes in the package `photoalbum.service`.

Things to notice about the code:

- The service gets its DAO through the constructor rather than creating one. `Main` creates each layer and hands it the layer below.
- The service doesn't know about HTTP. It throws `NotFoundException` and `ErrorHandler` decides what HTTP status code that is.
- The exception's message goes to the client, so it says only that there's no such photo. Saying "that photo belongs to another user" would tell the client that it exists — information that an attacker could use.

6 Determining the Current User — CurrentUser

Every photo album endpoint works with the current user's album. Normally the current user will be the one that has logged in, but for now the client will name the current user themselves in an `X-User` header attached to the HTTP request. This parallels what will happen when authentication is added — once logged in, the client will establish its identity using a (different) header.

Since (nearly) every route needs the current user's identity, a *before handler* — which is run before each route's own handler — is used to extract the information from the header and store it as an attribute of the request. Route handlers can then retrieve the current user's identity from the request attribute.

- Create `CurrentUser.java` as shown in listing 4 (in the appendix). It goes in the package `photoalbum.routes`.

Things to notice about the code:

- `register()` defines and registers the before handler to extract the current user information. `get()` is used by routes to retrieve the stored value.

- A missing or blank header throws `UnauthorizedException`.
- Anyone can send `X-User: arthur`, so for now any client can act as any user. Adding login (homework #13) fixes that and because routes only ever call `CurrentUser.get()`, the only code that will need to change is `register`.
- This version doesn't check that the named user exists. You'll add that once a few slices are working (see below).

7 The Route — PhotoRoutes

The route connects a URL to the service.

- Create `PhotoRoutes.java` as shown in listing 5 (in the appendix). It goes in the package `photoalbum.routes`.

Things to notice about the code:

- The handler does three things: read the request, call the service, write the response. It has no SQL and no rules.
- `pathId(ctx)` comes from `Params` (through the static import). It throws `ValidationException` for an id like `abc`.
- `ctx.json(photo)` turns the record into JSON. The status is 200 unless you set another one, and a null caption comes out as `"caption": null`.

8 Wire It Together — Main

Finally, `Main` (in package `photoalbum`) creates the layers, bottom up, and registers the handlers.

- Fill in the `TODOs` in `main` to create the DAO, service, and route objects and to register the routes. Also add a line to register the `CurrentUser` before handler. See listing 6 (in the appendix) for details.

You'll likely need to resolve a bunch of imports once you've added the new code — a quick way to do this in Eclipse is to right-click in the editor → **Source** → **Organize Imports**.

Things to notice about the code:

- Javalin always runs before handlers before the route handler so the order of the `register` steps doesn't matter, but registering `CurrentUser` first makes the code easier to understand because the order of statements in the code matches the order a request passes through them.

9 Run and Test

- Run Main as a Java application.

The console should show Javalin starting on port 7000. (When running the server again, be sure to first stop it with Eclipse’s red stop button, otherwise the run will fail because the port is already in use.)

- Once the application is running, try each of the following requests from a terminal — run one at a time. (The `-i` option makes `curl` show the status line and headers of the response as well as the body.) Think about what you expect the response to be for each (including the status code) before you run it — see the table of endpoints above.

```
# arthur asks for photo 1 (caption)
curl -i -H "X-User: arthur" http://localhost:7000/photos/1
# arthur asks for photo 4 (no caption)
curl -i -H "X-User: arthur" http://localhost:7000/photos/4
# ford asks for arthur's photo 1
curl -i -H "X-User: ford" http://localhost:7000/photos/1
# no X-User header
curl -i http://localhost:7000/photos/1
# non-numeric id
curl -i -H "X-User: arthur" http://localhost:7000/photos/abc
# no such photo (id doesn't exist)
curl -i -H "X-User: arthur" http://localhost:7000/photos/999999
```

If something goes wrong, the console often says why:

- A 500, with “Set DB_URL, DB_USER, and DB_PASSWORD” in the console: the environment variables aren’t set in your run configuration.
 - A 503: the database couldn’t be reached. Check the host and port in `DB_URL`. You need to be on campus or connected via the VPN to be able to access the server. (“bridgeman-vm.hws.edu: Name or service not known” is a sign that you are not within the campus network.)
 - A 404 for every request, even arthur’s own photo: check the resource name (path) in `PhotoRoutes.register()` and that Main calls `photoRoutes.register(config)`. If those are right, check that the photo exists in the database — connect to MySQL using MySQL Workbench and query the database directly.
 - A 500 with an `SQLException` in the console: usually a table or column name typo in the SQL.
-

The Remaining Slices

Build the rest of the required endpoints the same way: write a DAO method, a service method, a route handler, and one line in `PhotoRoutes.register()`, then run and test before going on. The following order is recommended so that each slice adds (only) one new idea.

10 GET /photos

This will be similar to the handling of `GET /books` in the library example, except that it is finding only the photos for the current user rather than all of the photos. Name the DAO method `findAll` and the service and route methods `listPhotos`. Both the DAO and service methods will need a `username` parameter; incorporate the username into the query as in `findById`.

An empty album should return status 200 with an empty list (`[]`), not a 404 (`NotFound`) error.

Testing:

```
# arthur's 17 photos
curl -i -H "X-User: arthur" http://localhost:7000/photos
# ford's 0 photos (empty album)
curl -i -H "X-User: ford" http://localhost:7000/photos
```

11 GET /photos?keyword=

This will be similar to the handling of `GET /books?author=` in the library example, again with the addition of a `username` parameter. Name the DAO method `findByKeyword` and the service method `listPhotosByKeyword`; update the already-implemented route method `listPhotos` to retrieve the keyword parameter and call the appropriate service method depending on whether or not it exists rather than implementing a new route method.

Testing:

```
# arthur's photos tagged 'waterfall'
curl -i -H "X-User: arthur" "http://localhost:7000/photos?keyword=waterfall"
# the same photos (MySQL compares the strings without regard to case)
curl -i -H "X-User: arthur" "http://localhost:7000/photos?keyword=Waterfall"
# arthur's photos tagged 'New Zealand' (%20 is a space in a URL)
curl -i -H "X-User: arthur" "http://localhost:7000/photos?keyword=New%20Zealand"
# arthur's photos tagged 'penguin' (no photos)
curl -i -H "X-User: arthur" "http://localhost:7000/photos?keyword=penguin"
# ford has no photos at all
curl -i -H "X-User: ford" "http://localhost:7000/photos?keyword=waterfall"
```

12 POST /photos

This will be similar to the handling of POST /books in the library example with the addition of a username — a new photo is added for the current user. Name the DAO method `insert` and the service and route methods `addPhoto`. Remember to include a `username` parameter for the DAO and service methods.

You will need a DTO `NewPhoto`, similar to `NewBook`, to capture the information sent by the client (a url and, optionally, a caption). `NewPhoto` should go in the package `photoalbum.dto`.

The validation rules are given in the Required Endpoints table — look for the reasons for a 400 (Bad Request) error status code. URLs should be trimmed since spaces are not legal characters. (Any actual space must be escaped with `%20`.)

Testing:

```
# add a new photo for arthur (without caption)
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_4881.JPG"}' \
  http://localhost:7000/photos

# attempt to add a duplicate url for arthur
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_4881.JPG"}' \
  http://localhost:7000/photos

# add a new photo for ford (the same as one of arthur's)
curl -i -X POST -H "X-User: ford" -H "Content-Type: application/json" \
  -d '{"url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_4881.JPG"}' \
  http://localhost:7000/photos

# add a new photo for arthur (with caption)
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_0176-edit.JPG",
    "caption": "a misty day and waterfalls in Doubtful Sound"}' \
  http://localhost:7000/photos

# no URL
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"caption": "No URL"}' \
  http://localhost:7000/photos

# not an http:// or https:// URL
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"url": "ftp://example.com/a.jpg"}' \
  http://localhost:7000/photos

# not valid JSON
curl -i -X POST -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"url": ' \
  http://localhost:7000/photos
```

Verify the additions using Workbench to examine the `ALBUM` table or by sending requests to retrieve the new photos.

13 PUT `/photos/{id}/caption`

This will be similar to the handling of `PUT /books/{id}` in the library example, except that only one column changes and it only applies to the current user. Name the DAO method `updateCaption` and the service and route methods `setCaption`. Remember to include a `username` parameter for the DAO and service methods.

Create a `NewCaption` DTO for the caption-only info being sent from the client. A missing caption field and `"caption": null` will both arrive as `null` in the record.

The validation rules are given in the Required Endpoints table — look for the reasons for a 400 (Bad Request) error status code. Also notice that the description of the endpoint says that `null` or `""` clears the caption (set it to `null`).

Testing:

```
# add a caption
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"caption": "kayaks on Georgian Bay"}' \
  http://localhost:7000/photos/4/caption

# clear caption (null)
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"caption": null}' \
  http://localhost:7000/photos/4/caption

# clear caption ("")
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"caption": ""}' \
  http://localhost:7000/photos/4/caption

# ford can't caption arthur's photo
curl -i -X PUT -H "X-User: ford" -H "Content-Type: application/json" \
  -d '{"caption": "Mine"}' \
  http://localhost:7000/photos/4/caption

# no such photo
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{"caption": "Nope"}' \
  http://localhost:7000/photos/999999/caption

# missing caption field (clears the caption, like null)
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  -d '{}' http://localhost:7000/photos/4/caption

# no body at all (not valid JSON)
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
  http://localhost:7000/photos/4/caption

# a 201-character caption (the shell builds the string of x's)
curl -i -X PUT -H "X-User: arthur" -H "Content-Type: application/json" \
```

```
-d "{\"caption\": \"$(printf '%.0s' {1..201})\"}" \
http://localhost:7000/photos/4/caption
```

Verify the changes using Workbench to examine the `ALBUM` table or by sending requests to retrieve the modified information.

14 DELETE /photos/{id}

This will be similar to the handling of `DELETE /books/{id}` in the library example except for dealing with a photo's keywords — because `ON DELETE CASCADE` is set for `KEYWORD`'s foreign key, removing a photo with keywords automatically removes its keywords instead of resulting in an error. As with the other operations, delete only applies to the current user's photos.

Name the DAO method `delete` and the service and route methods `deletePhoto`. Remember to include a `username` parameter for the DAO and service methods.

Testing:

```
# ford can't delete arthur's photo
curl -i -X DELETE -H "X-User: ford" http://localhost:7000/photos/17
# no body (valid delete operation)
curl -i -X DELETE -H "X-User: arthur" http://localhost:7000/photos/17
# deleting it again
curl -i -X DELETE -H "X-User: arthur" http://localhost:7000/photos/17
```

Verify the deletions using Workbench to examine the `ALBUM` table or by sending requests attempting to retrieve the deleted information.

15 401 For Nonexistent User

`401 Unauthorized` (meaning a not logged in user) is the correct response for an `X-User` that names a user who doesn't exist — this will correspond to the client not providing proper authentication credentials once authentication is added.

Implementing this means checking that the user exists in `CurrentUser.register`, however `CurrentUser` is a route and should neither contain SQL (in order to retrieve information from the `USER` table) nor access a DAO directly. Instead:

- Create `UserDao` with a method `exists` that reports whether a username exists (`SELECT 1 FROM USER WHERE username = ?`, then check whether the result set has a row).
- Create `UserService` with a method `requireUser` that uses `UserDao.exists` to determine if the user exists and throws `UnauthorizedException` if not. This method is void — it either throws an exception or returns quietly.

- `CurrentUser.register` needs that service, so `CurrentUser` becomes an object that's given a `UserService` in its constructor, the same way `PhotoRoutes` is given `PhotoService`. Make the constructor public and `register` an instance method (remove `static`) that calls the service after checking that the header is present. `get` can stay `static`, since it only reads the request attribute, so no route has to change.
- In `Main`, create the `UserDao`, `UserService`, and `CurrentUser` alongside the photo objects, and call `currentUser.register(config)` in place of `CurrentUser.register(config)`.

Testing:

```
# there's no user named zaphod
curl -i -H "X-User: zaphod" http://localhost:7000/photos
# no X-User header at all
curl -i http://localhost:7000/photos
# arthur and ford still work - retrieve their photos
curl -i -H "X-User: arthur" http://localhost:7000/photos
curl -i -H "X-User: ford" http://localhost:7000/photos
```

16 GET /photos/{id}/keywords

This combines two slices you've already built: like `GET /photos/{id}`, it starts by finding one of the current user's photos, and like `GET /photos`, it returns a list (which may be empty).

- Put the SQL for the `KEYWORD` table in a new `KeywordDao`. The `findByPhoto` method should take a photo id and return a `List<String>` of keywords (`SELECT keyword FROM KEYWORD WHERE id = ? ORDER BY keyword`) — similar to how `PhotoDao.findAll` works.
- Add a `KeywordDao` as a second constructor parameter for `PhotoService`.
- Add a method `listKeywords` to `PhotoService` to return the keywords for the photo. Since you need different behavior for “no such photo” vs just “no keywords” and the keyword query alone can't distinguish these cases (both result in an empty list), first check the photo by calling `getPhoto(id, username)` before fetching the keywords. `getPhoto` throws `NotFoundException` for a photo that's missing or not the user's, handling the “no such photo” case. (Note that you don't need to do anything with the `Photo` returned by `getPhoto` — you simply want it to either throw the exception or not.) Also note that since the ownership check has been taken care of by `getPhoto`, the keyword query doesn't need the username.
- Put the route in `PhotoRoutes` — add a `listKeywords` method (and register the route).
- `Main` creates the new objects.

- `ctx.json` turns a `List<String>` into a JSON array of strings, such as `["mountain", "New Zealand"]`.

Testing:

```
# get photo 1's keywords
curl -i -H "X-User: arthur" http://localhost:7000/photos/1/keywords
# get photo 18's keywords (no keywords) - photo 18 was added in an earlier test
# change to a different photo id as needed if add/delete tests have been re-run
curl -i -H "X-User: arthur" http://localhost:7000/photos/18/keywords
# not ford's photo
curl -i -H "X-User: ford" http://localhost:7000/photos/1/keywords
# no such photo
curl -i -H "X-User: arthur" http://localhost:7000/photos/999999/keywords
```

17 (Optional) POST /photos/{id}/keywords

This combines two things you’ve already done. Check the photo the same way as for GET /photos/{id}/keywords: the service calls `getPhoto` first, and only if that succeeds does it call a `KeywordDao` method that does the INSERT. The request body and the 201 response work like POST /photos. Things to watch for:

- Trim the keyword before storing it. Keywords are compared with each other and " volcano" (with a leading space) would otherwise be a different keyword from "volcano".
- A keyword the photo already has violates `KEYWORD`’s primary key, which `SqlErrors` already turns into a 409. MySQL compares the strings without regard to case, so “Volcano” is a duplicate of “volcano.” The message `SqlErrors` produces is generic; the service can catch the `ConflictException` and throw a new one that says which keyword was the duplicate.
- If the photo is deleted between the check and the insert, the insert violates the foreign key and `SqlErrors` turns that into a 404, so this case is taken care of without any additional code.
- The `getPhoto` check is necessary — if the photo doesn’t exist, the insert violates the foreign key and `SqlErrors` turns that into a 404 as desired, but the foreign key can’t stop a user from adding a keyword to someone else’s photo.

18 (Optional) GET /keywords

This is similar to GET /photos. Things to watch for:

- Make sure you only include keywords for the current user's photos. (Also note that the sample data provided won't reveal a mistake here, since only arthur has any keywords.)
- Many photos share a keyword, and each keyword should appear in the list only once, in alphabetical order.
- MySQL compares strings without regard to case, so if "Lake" is on one photo and "lake" on another, removing the repeats counts them as the same keyword, and you'll get back one of the two spellings. That's fine here, but notice that it happens.

19 (Optional) PUT /photos/{id}/keywords

The ownership check and the validation of each keyword work like POST /photos/{id}/keywords and the response can reuse the service method behind GET /photos/{id}/keywords.

Things to watch for:

- The body of the request is {"keywords": [...]} — there's a list of keywords. Create a DTO record to capture this:

```
public record KeywordsRequest(List<String> keywords) {  
}
```

`request.keywords()` returns a `List<String>`. A missing `keywords` field (a 400) means a `null` return value; an empty list is valid and removes all of the photo's keywords.

- Drop duplicates within the list, ignoring case, since the database will — a list with both "Lake" and "lake" would otherwise fail partway through the inserts. A `TreeSet` created with `String.CASE_INSENSITIVE_ORDER` keeps only one spelling of each.

```
Set<String> set = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
```

- Validate every keyword before changing anything so bad input never reaches the database.

There is an additional problem: if the database fails partway through (after the delete but before all of the inserts), the photo is left with its old keywords gone and only some of the new ones added. Making a group of statements succeed or fail as a unit takes a *transaction*, a topic coming soon.

Notice also that sending the same request twice leaves the photo in the same state as

sending it once. That's what makes PUT the right method here, rather than POST.

Appendix

This source code can also be found at <http://math.hws.edu/bridgeman/courses/343/f26/hw/photoalbum> if you have trouble copy-and-pasting from this document.

```
package photoalbum.domain;

/**
 * One photo in a user's album: a row of the ALBUM table.
 *
 * id is null for a photo that hasn't been inserted yet. caption may be null
 * (nullable columns). The owner (ALBUM.username) isn't stored here: the DAO
 * only returns photos belonging to the current user, so every Photo already has
 * a known owner.
 */
public record Photo(Integer id, String url, String caption) {
}
```

Listing 1: Photo.java

```
package photoalbum.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Optional;

import photoalbum.db.DB;
import photoalbum.domain.Photo;

/**
 * All of the SQL for the ALBUM table.
 *
 * Every lookup includes the owner, so a photo that belongs to someone else is
 * indistinguishable from one that doesn't exist.
 */
public class PhotoDao {
    // The photo with this id in this user's album, or empty if there
    // is none.
    public Optional<Photo> findById(int id, String username) {
        String sql = "SELECT id, url, caption FROM ALBUM WHERE id = ? " +
            "AND username = ?";
        try (Connection conn = DB.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setInt(1, id);
            stmt.setString(2, username);
            try (ResultSet rs = stmt.executeQuery()) {
                if (rs.next()) {
                    return Optional.of(toPhoto(rs));
                }
                return Optional.empty();
            }
        } catch (SQLException e) {
            throw SqlErrors.translate(e);
        }
    }

    // The current row of rs as a Photo. getString returns null for a
    // NULL caption.
    private Photo toPhoto(ResultSet rs) throws SQLException {
        return new Photo(rs.getInt("id"), rs.getString("url"), rs.getString("caption"));
    }
}
```

Listing 2: PhotoDao.java

```
package photoalbum.service;

import java.util.Optional;

import photoalbum.dao.PhotoDao;
import photoalbum.domain.Photo;
import photoalbum.errors.NotFoundException;

/**
 * The rules for working with photos.
 *
 * Knows nothing about HTTP: it takes plain values and returns domain objects,
 * and reports failures by throwing the application's exceptions.
 */
public class PhotoService {

    private final PhotoDao dao;

    public PhotoService(PhotoDao dao) {
        this.dao = dao;
    }

    // One of username's photos. NotFoundException if there's no such
    // photo or it belongs to another user.
    public Photo getPhoto(int id, String username) {
        Optional<Photo> photo = dao.findById(id, username);
        if (photo.isEmpty()) {
            throw new NotFoundException("No photo with id " + id);
        }
        return photo.get();
    }
}
```

Listing 3: PhotoService.java

```
package photoalbum.routes;

import io.javalin.config.JavalinConfig;
import io.javalin.http.Context;
import photoalbum.errors.UnauthorizedException;

/**
 * Works out whose album a request is for.
 *
 * For now the client simply names the user in an X-User header. Once login is
 * added, only register() changes (to read a token instead); routes keep calling
 * CurrentUser.get(ctx).
 */
public final class CurrentUser {

    private static final String KEY = "currentUser";

    private CurrentUser() {}

    // Set up a before handler to extract the current user from the
    // request.
    // Called from inside Javalin.create(config -> { ... })
    public static void register(JavalinConfig config) {
        // beforeMatched runs before every request that matches a route
        // (an unknown path still gets a 404, not a 401)
        config.routes.beforeMatched(ctx -> {
            String username = ctx.header("X-User");
            if (username == null || username.isBlank()) {
                throw new UnauthorizedException("X-User header is required");
            }
            ctx.attribute(KEY, username.trim());
        });
    }

    // The current user's username, as stored by the before handler.
    public static String get(Context ctx) {
        return ctx.attribute(KEY);
    }
}
```

Listing 4: CurrentUser.java

```
package photoalbum.routes;

import static photoalbum.routes.Params.pathId;

import io.javalin.config.JavalinConfig;
import io.javalin.http.Context;
import photoalbum.domain.Photo;
import photoalbum.service.PhotoService;

/**
 * The HTTP side of the photo endpoints.
 *
 * Each handler reads the request, calls the service, and writes the response.
 * No SQL and no rules here; failures are exceptions that ErrorHandler turns
 * into status codes.
 */
public class PhotoRoutes {

    private final PhotoService service;

    public PhotoRoutes(PhotoService service) {
        this.service = service;
    }

    // Called from inside Javalin.create(config -> { ... })
    public void register(JavalinConfig config) {
        config.routes.get("/photos/{id}", ctx -> getPhoto(ctx));
    }

    // GET /photos/{id} -> 200 and the photo
    private void getPhoto(Context ctx) {
        int id = pathId(ctx);
        String username = CurrentUser.get(ctx);
        Photo photo = service.getPhoto(id, username);
        ctx.json(photo);
    }
}
```

Listing 5: PhotoRoutes.java

```
package photoalbum;

// ...imports omitted for space reasons...

/**
 * Starts the photo album API on port 7000.
 *
 * The only class that knows every layer: it builds them bottom-up and hands
 * each one the layer below.
 */
public class Main {

    public static void main(String[] args) {
        // TODO: create DAOs
        PhotoDao photoDao = new PhotoDao();

        // TODO: create services
        PhotoService photoService = new PhotoService(photoDao);

        // TODO: create routes
        PhotoRoutes photoRoutes = new PhotoRoutes(photoService);

        // Javalin 7: all configuration and routes go inside the config block
        Javalin.create(config -> {
            config.jetty.port = 7000;

            // JSON: dates as "1979-10-12" rather than [1979,10,12]
            // and ignore JSON fields a DTO doesn't have
            config.jsonMapper(new JavalinJackson().updateMapper(mapper -> {
                mapper.registerModule(new JavaTimeModule());
                mapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
                mapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
            }));

            // One place turns every exception into a status code and error JSON
            ErrorHandler.register(config);

            // Every route needs to know whose album the request is for
            CurrentUser.register(config);

            // TODO: register routes
            photoRoutes.register(config);

            // TODO: register routes
        }).start();
    }
}
```

Listing 6: Main.java