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-13a` ***for your first handin and*** `photoalbum-13b` ***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 11/9 and Monday 11/16.

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.

In homework #12, each request simply named the user whose album it worked with, in an `X-User` header, so anyone could view or modify anyone's album just by claiming to be someone else. This homework adds authentication — users can create an account, log in, and log out and every other request must show that it comes from a logged-in user. The current user is then whoever is logged in so a user can only access their own album. This homework also adds a role: administrators, who can list all of the users and delete any account.

The username and password associated with each user of the photo management system are distinct from the username and password your application uses to connect to the MySQL server. MySQL verifies the application's database password and enforces that account's permissions. For the photo album's users, though, the database only stores the information: a username and a hash of the password in `USER`. It is up to the application to verify each user's password when they log in, and to enforce which albums each user may view or change.

You'll continue working in the same `photoalbum` project you used for homework #12; this homework builds on that code. You should complete homework #12 first.

Database

The database is the same as in homework #12. Continue to use your MySQL account rather than `guest` for the application's connection to the database.

The sample users are `arthur` (password `tea`) and `ford` (password `betelgeuse`).

The database schema:

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

`USER`, `ALBUM`, and `KEYWORD` are the same as in homework #12. In this homework you'll be working with `USER.password`, which holds a bcrypt *hash* of the user's password rather than the password itself. Not used in homework #12 is `SESSION`, which supports session-based authentication. It holds one row for each login containing a random token, the user it belongs to, and when it expires.

Foreign keys:

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

All foreign key constraints are `ON DELETE CASCADE`. In addition:

- `(ALBUM.username,ALBUM.url)` is `UNIQUE`.
- All columns except `ALBUM.caption` are `NOT NULL`.

Endpoints

This homework adds five new required endpoints and one optional endpoint, listed in the next two pages.

Conventions that apply to every endpoint:

- Request and response bodies are JSON. Every error response has the same format: `{"status": 401, "message": "..."}`.
- Every endpoint except `POST /users` and `POST /login` requires the header `Authorization: Bearer` *`token`*, where *token* is the token returned by `POST /login`. The current user is the user the token belongs to. A missing, unknown, or expired token produces a `401` response. (The `X-User` header is no longer used.)
- A request from a logged-in user who isn't allowed to do what they asked produces a `403` response.
- A login session lasts 8 hours.
- Any unexpected failure produces a `500` response. This is not listed separately below.

API Implementation

As in homework #12, new features will be built in vertical slices — one endpoint at a time rather than layer by layer.

The "library demo" referred to throughout is the "with authorization and authentication" version from 10/26 — find the code posted on the schedule page.

Required Endpoints

Endpoint	Description	Request	Success response	Errors
`POST /users`	Create an account.	Body: `{"username": "trillian", "password": "improbable"}`	`201` `{"username": "trillian"}`	`400` malformed JSON; username missing, blank, or over 50 characters; password missing or under 8 characters `409` username already taken
`POST /login`	Log in and receive a session token.	Body: `{"username": "arthur", "password": "tea"}`	`200` `{"token": "Xq3v...9sQ"}`	`400` malformed JSON `401` invalid username or password (including a missing one)
`POST /logout`	End the current session; its token no longer works.	None (the token in the `Authorization` header)	`204` (no body)	`401`
`GET /users`	List all users, alphabetically. Administrators only.	None	`200` `[{"username": "arthur"}, {"username": "ford"}, ...]`	`401` `403` not an administrator
`DELETE /users/{username}`	Delete an account, along with its photos, keywords, and sessions. Users can delete their own account; administrators can delete any account.	Path: `username`	`204` (no body)	`401` `403` someone else's account, and not an administrator `404` no such user (administrators only; anyone else gets `403`)

Optional Endpoints

Endpoint	Description	Request	Success response	Errors
`GET /users/me`	Who is the current user?	None	`200 {"username": "arthur", "admin": false}`	`401`

`POST /users`

`POST /users` creates an account. For example, the request

```
POST /users
Content-Type: application/json

{"username": "trillian", "password": "improbable"}
```

adds the user `trillian` and produces a `201` response whose body describes the new account:

```
{"username": "trillian"}
```

This is the photo album's version of the library's `POST /members` (register).

1 The Domain Object — `User`

A `User` holds one row of the `USER` table.

- Create `User.java` as shown in listing 1 (in the appendix). It goes in package `photoalbum.domain`. Compare it to the library's `domain/Member.java`.

Things to notice about the code:

- Unlike `Member`, there's no id (`USER`'s primary key is the username itself) and no role (administrators are configured separately; see Administrators, below).
- `User` includes the password hash because logging in (the next slice) has to check a password against it. That's also why a `User` must never be sent to the client.

2 The DTOs — `NewUser` and `UserView`

DTOs describe what goes between the client and the API.

- Create `NewUser.java` and `UserView.java` as shown in listings 2 and 3 (in the appendix). They go in package `photoalbum.dto`. Compare to the library demo's `dto/NewMember.java` and `dto/MemberView.java`.

Things to notice about the code:

- `NewUser` lists the only fields a client can supply when registering. Because `Main` tells Jackson to ignore fields a DTO doesn't have, anything else in the request body is dropped. (In the library demo, this is what stops a client from registering with `"role": "librarian"`.)
- `NewMember` leaves out the fields of `Member` that the client doesn't supply (id and role). `User` has no such fields, so `NewUser` looks very similar to `User` and might

seem unnecessary — but it is important to note that `User`, like the `USER` table, actually stores a *password hash*, not a plain password, while the client sends the plain password.

- `UserView` is what the API sends back. It's separate from `User` so that the password hash can't end up in a response by accident. `from` makes one from a `User`, like `MemberView.from`.

3 The DAO — `UserDao`

`UserDao` was created in homework #12 with a method to determine if a particular user exists. Now add methods to add a new user and to find users by username.

- Add `findByUsername` and `insert` methods (as shown in listing 4) to your `UserDao`. Compare to `insert` and `findByUsername` in the library's `dao/MemberDao.java`.

Things to notice about the code:

- There's no generated key as in the library because the username is the primary key instead of `member_id`, so `insert` doesn't use `RETURN_GENERATED_KEYS` and doesn't return anything.
- Unlike the library's DAO methods, these catch the `SQLException` and throw `SqlErrors.translate(e)` instead, as `PhotoDao` does.
- Registering a username that's already taken violates the primary key, which `SqlErrors` turns into a `ConflictException` (`409`). Note that MySQL compares strings without regard to case, so `Trillian` is a duplicate of `trillian`.
- `insert` is handed a hash, never the plaintext password.
- `findByUsername` is used by the service to read back the new user. (Other services also make use of it.) `toUser` creates a new `User` from a result row; the library's version is named `map`. `PhotoDao.toPhoto` plays a similar role.

4 The Service — `UserService`

The service checks the request and hashes the password.

- Add the code shown in listing 5 to your `UserService`. When resolving imports, be sure to get the `ValidationException` from `photoalbum.errors`. Compare with `register` in the library's `service/MemberService.java`.

Things to notice about the code:

- The validation throws `ValidationException` (a `400`) instead of the library's `IllegalArgumentExcepti` as in `PhotoService`.

- Validation is important. A username over 50 characters would otherwise reach MySQL, whose "data too long" error isn't recognized by `SqlErrors` so it would come back as an uninformative `500`. The minimum password length is a policy constraint not present in the library demo — 8 characters has a long history, though on modern systems that is too short and 12-16+ characters are recommended. Note that while the username max length is backed up by the database, the minimum password length is only checked by the application at the point of account creation — so arthur's existing very short password, created before the 8 character policy was adopted, is allowed.
- The username is trimmed, since it's an identifier. The password isn't: spaces are allowed in a password.
- `BCrypt.hashpw(password, BCrypt.gensalt())` hashes the password, exactly as in the library demo. `gensalt` makes a new random salt each time so two users with the same password get different hashes.
- The `ConflictException` from `SqlErrors` just says "Duplicate value", so the service catches it and throws one with a clearer message.
- Like the library demo, the service reads the new user back after inserting it. The library did this to get the member ID the database generated. Here nothing is generated, since the username is the primary key, but reading the user back still means the response shows what was actually stored. If it isn't there, something has gone badly wrong, so the service throws an `AppException` (a `500`.)

5 The Route — `UserRoutes`

The route connects a URL to the service.

- Create `UserRoutes.java` as shown in listing 6. It goes in package `photoalbum.routes`. Compare to `register` and `create` in the library's `routes/MemberRoutes.java`.

Things to notice about the code:

- The handler reads the body as a `NewUser`, calls the service, and responds with `201` and the new user as a `UserView`, never a `User`.
- Unlike the demo's `create`, it has no `try`/`catch`: a `ValidationException` goes to `ErrorHandler` like any other exception.
- There's no `Location` header, since the API has no URL for getting a particular user. (REST convention is to respond with the URL a client could use to retrieve the newly created item, but the photoalbum API doesn't include one.)
- The handler is called `createUser` because `register` is already the name of the method that registers the routes.

6 Make It Public — `CurrentUser`

Every route in homework #12 acts on the current user's photos, so they (currently) require an `X-User` header to identify the user. `POST /users`, however, has to work for someone who doesn't have an account yet.

- Add `isPublic` to `CurrentUser` as shown in listing 7. Compare to `isPublic` in the library's `routes/AuthFilter.java`.
- Call `isPublic` as the first step in the before handler, so that `register` in `CurrentUser` is as shown in listing 7.

Things to notice about the code:

- Everything is protected unless it's on the list in `isPublic`. Forgetting to list a public route makes it unusable, which you'll notice right away. Forgetting to protect a route would leave it open, which you might not notice.
- The check uses the method as well as the path. `GET /users` has the same path as `POST /users`, but it isn't public.

7 Wire It Together — `Main`

- In `Main`, create a `UserRoutes` (given your existing `UserService`) and register it, similar to what is done for `PhotoRoutes` and what the library's `Main` does for `MemberRoutes`.

8 Run and Test

```
# create an account (no X-User header needed)
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "trillian", "password": "improbable"}' \
     http://localhost:7000/users
# the same username again
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "trillian", "password": "xyzzy123"}' \
     http://localhost:7000/users
# MySQL compares strings without regard to case
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "Trillian", "password": "improbable"}' \
     http://localhost:7000/users
# password too short
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "marvin", "password": "glum"}' \
     http://localhost:7000/users
# no password
curl -i -X POST -H "Content-Type: application/json" \
```

```
     -d '{"username": "marvin"}' \
     http://localhost:7000/users
# no username
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"password": "improbable"}' \
     http://localhost:7000/users
```

Check the `USER` table in Workbench: the `password` values for the new rows should be 60-character hashes beginning with `$2a$`, not the passwords themselves.

`POST /login`

`POST /login` checks a username and password and, if they're correct, starts a session. For example, the request

```
POST /login
Content-Type: application/json

{"username": "arthur", "password": "tea"}
```

produces a `200` response whose body is a new session token:

```
{"token": "Xq3vH0dJ...9sQ"}
```

The client sends this token with every later request (the slice after this one makes that work).

This is the photo album's version of the library's `POST /login`.

9 The Domain Object

The `SESSION` table is purely implementation — there's no notion of a session in the photo album domain. As a result, no domain object is needed.

10 The DTOs — `LoginRequest` and `TokenResponse`

These capture what the client sends to log in (username and password — plaintext password, not a password hash) and the token returned after a successful login. The code was provided in the homework #12 starter code so there's nothing to write here, just look at the two records (in the `photoalbum.dto` package).

11 The DAO — `SessionDao`

`SessionDao` contains the SQL for accessing the `SESSION` table.

- Create `SessionDao.java` as shown in listing 8. Put it in package `photoalbum.dao`. Compare to the library demo's `dao/SessionDao.java`.

Things to notice about the code:

- `SESSION` refers to the user by username; the library's `SESSIONS` uses a numeric member id.
- `setObject` with a `LocalDateTime` stores a `DATETIME`, as in the demo.
- The library's `SessionDao` has additional methods for looking up and deleting a session. You'll add your versions of them in later slices.

12 The Session Service — `SessionService`

The service creates a token and starts a session.

- Create `SessionService.java` as shown in listing 9. Put it in the package `photoalbum.service`. Compare to `start` in the library's `service/SessionService.java`.

Things to notice about the code:

- A token is 32 random bytes, written as text with Base64 (43 characters). `SecureRandom` generates numbers that can't be predicted from earlier ones; `Random` doesn't and predictable tokens could be guessed.
- The token is the only thing that identifies a logged-in user, so anyone who has it can act as that user until it expires (8 hours, as in the library demo) or is ended by logging out.
- The session is stored in the database rather than in the server's memory so restarting the server doesn't log anyone out.

13 The Service — `UserService`

The service authenticates the user.

- `UserService` needs a `SessionService` to start sessions. Replace its instance variable and constructor with those shown in listing 10. Compare to the constructor in the library's `service/MemberService.java`.
- Add the `login` method as shown in listing 10. Compare to `login` in the library's `service/MemberService.java`.

Things to notice about the code:

- `BCrypt.checkpw` hashes the password the client sent using the salt stored in the hash and compares the result with the stored hash. (The stored hash can't be turned back into the password.)
- An unknown username and a wrong password get exactly the same response. If they differed, anyone could find out which usernames exist by trying to log in.
- In the library demo, `login` returns an empty `Optional` for a failed login and the route turns that into a `401`. Here the service throws `UnauthorizedException` itself so the route doesn't need to check anything.
- The session is started for the `User` found in the database, not the username the client typed. MySQL compares without regard to case, so logging in as `ARTHUR` finds arthur's row. The session belongs to `arthur`, as stored.
- A missing username or password is just an invalid login (`401`), as in the library demo.

14 The Route — `AuthRoutes`

The route connects a URL to the service.

- Create `AuthRoutes.java` as shown in listing 11. Put it in package `photoalbum.routes`. Compare to `register` and `login` in the library's `routes/AuthRoutes.java`.

Things to notice about the code:

- The handler reads the body as a `LoginRequest`, calls the service, and sends the token as a `TokenResponse`, with the default status of `200`.
- The library's `AuthRoutes` has additional methods which will be added later.

15 Make It Public — `CurrentUser`

Users also need to be able to log in without already being logged in — `POST /login` needs to be a public route.

- Add `POST /login` to the public routes: in `isPublic` in `CurrentUser`, `return path.equals("/users") || path.equals("/login")`.

16 Wire It Together — `Main`

- Update `Main` to create the new pieces, give the `SessionService` to `UserService`, and register the new routes:

```
// DAOs
SessionDao sessionDao = new SessionDao();
...
// services: SessionService first, since UserService needs it
SessionService sessionService = new SessionService(sessionDao);
UserService userService = new UserService(userDao, sessionService);
...
// routes
AuthRoutes authRoutes = new AuthRoutes(userService);
...
// inside Javalin.create(config -> { ... }), with the other routes
authRoutes.register(config);
```

17 Run and Test

```
# log in as arthur; the response has a token
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "arthur", "password": "tea"}' \
     http://localhost:7000/login
# wrong password
```

```
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "arthur", "password": "coffee"}' \
     http://localhost:7000/login
# no such user (compare the response to the previous one)
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "zaphod", "password": "tea"}' \
     http://localhost:7000/login
# the account created in the previous slice
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "trillian", "password": "improbable"}' \
     http://localhost:7000/login
# the username in a different case
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "ARTHUR", "password": "tea"}' \
     http://localhost:7000/login
```

Each successful login adds a row to `SESSION`; check them in Workbench. The row for the last login should have the username `arthur`, not `ARTHUR`. Also note that repeated logins add additional sessions instead of replacing the current one.

Tokens Instead of `X-User`

This slice switches the whole API from `X-User` to tokens: every request (except registering and logging in) must now include the header

```
Authorization: Bearer token
```

where *token* is a token returned by `POST /login`. The current user will be the user that token belongs to — this must be looked up in the `SESSION` table.

18 The DAO — `SessionDao`

- Add the `findUsername` method shown in listing 12 to your `SessionDao`. Compare to `findMember` in the library's `dao/SessionDao.java`.

Things to notice about the code:

- `expires_at > ?` means an expired session is treated exactly like a token that doesn't exist. Expired rows stay in the table; they're just never matched.
- The library demo joins `MEMBERS` to get the whole member, including the role. `SESSION` already has the username, which is all the photo album needs, so there's no join.

19 The Session Service — `SessionService`

Compare the following to `memberForToken` in the library's `service/SessionService.java`.

- Add this method to your `SessionService`:

```
// The username a token belongs to, if the token is valid.
public Optional<String> usernameForToken(String token) {
  return dao.findUsername(token);
}
```

Things to notice about the code:

- It just passes the call along to the DAO. That's fine: there's no rule to apply and routes still go through the service rather than using a DAO directly.

20 Check the Token — `CurrentUser`

`CurrentUser` now needs to check for a token and use that for the current user instead of relying on the `X-User` header.

- Replace your `CurrentUser` with the version shown in listing 13. It does the same job as before — determining the current user and storing that information in the request `Context` — but now does it by getting the token from the `Authorization: Bearer` header and looking up the associated user in the `SESSION` table. Compare to `handle` in the library's `routes/AuthFilter.java` and `bearerToken` in `routes/Auth.java`.

Things to notice about the code:

- `CurrentUser` is now given a `SessionService` instead of the `UserService` it used for `X-User` so it can look up the token.
- `isPublic` is unchanged since the `POST /users` slice.
- The before handler is the same as the library's `AuthFilter.handle` except for throwing `UnauthorizedException` where the demo throws `UnauthorizedResponse`. A missing header, an unknown token, and an expired token all get the same `401`.
- `bearerToken` comes from the library's `Auth` class.
- `CurrentUser` does the same job as `AuthFilter` and `Auth` in the library demo — compare the two versions to see how they line up. Both are reasonable designs: the single-class version is simpler for small projects like the photo album, while splitting the stateless static helpers that any route can use from the part that needs a service (`Auth` vs `AuthFilter`) scales better when there are more roles and more helpers.
- The library's filter stores the whole `Member` instead of just the username. The library's approach is better when roles are stored in the database because checking an authenticated user's role doesn't require another lookup — it is right there in `Member`. For the photo album, which doesn't store roles in the database, the username-only version is simpler and is compatible with the design from homework #12.

21 Remove the `X-User` Check

- Remove `requireUser` from `UserService` and `exists` from `UserDao`.

Things to notice about the code:

- These methods aren't needed any more: `SESSION`'s foreign key guarantees that a session belongs to a user who exists, so finding a valid session is enough to know that the user exists.

22 Wire It Together — `Main`

- `CurrentUser` now needs the `SessionService`. Change where `Main` creates it (after the `SessionService` has been created):

```
CurrentUser currentUser = new CurrentUser(sessionService);
```

23 Run and Test

Ownership checks are unchanged: your queries already include `AND username = ?`. What's different is that the username now comes from a login instead of from whatever the client typed into `X-User`, so those checks now actually protect each user's album.

To facilitate testing, save the token you get on login in a shell variable. In a terminal window:

```
# log in as arthur; the response has a token
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "arthur", "password": "tea"}' \
     http://localhost:7000/login

# store token - paste from the login response as indicated
ARTHUR=paste-arthurs-token-here
```

Do the same for ford, storing the token in a shell variable `FORD`. The shell variables will persist until you close the terminal. To change the value of the stored token, repeat the assignment with the new value. (Note that because the sessions are stored in a database table, logins persist across server restarts — you don't have to log in again and get new tokens unless the current token is no longer valid because it has expired or the user logged out.)

Then rerun your homework #12 tests, replacing

```
-H "X-User: arthur"
```

with

```
-H "Authorization: Bearer $ARTHUR"
```

(and likewise for ford), and check that each gives the same result as before. Also try:

```
# no token
curl -i http://localhost:7000/photos
# X-User no longer works
curl -i -H "X-User: arthur" http://localhost:7000/photos
# a made-up token
curl -i -H "Authorization: Bearer not-a-real-token" \
     http://localhost:7000/photos
# arthur's photos, using arthur's token
curl -i -H "Authorization: Bearer $ARTHUR" http://localhost:7000/photos
# ford's token can't reach arthur's photo
curl -i -H "Authorization: Bearer $FORD" http://localhost:7000/photos/1
# logging in still works without a token
```

```
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "arthur", "password": "tea"}' \
     http://localhost:7000/login
# previous token still works - repeat logins don't invalidate existing tokens
curl -i -H "Authorization: Bearer $ARTHUR" http://localhost:7000/photos
```

To see an expired session rejected, change a token's `expires_at` in Workbench to a time in the past and try the token again.

`POST /logout`

`POST /logout` ends the session whose token was sent so that the token stops working. It has no body; a successful logout is a `204` with no response body.

This is the photo album's version of the demo's `POST /logout`.

24 The DAO — `SessionDao`

- Add the `delete` method shown in listing 12 to your `SessionDao`. Compare to `delete` in the library's `dao/SessionDao.java`.

25 The Session Service — `SessionService`

- Add the `end` method shown in listing 15 to your `SessionService`. Compare to `end` in the library's `service/SessionService.java`.

Things to notice about the code:

- `end` ignores the return value indicating whether or not a row was actually deleted. By the time it is called, the before handler has already confirmed the token is valid so we know there's a row to delete.

26 The Route — `AuthRoutes`

- Update your `AuthRoutes` to match the version in listing 16, which adds the `SessionService`, the `POST /logout` route, and its handler. Compare to your `AuthRoutes` before replacing it and compare the result to the constructor, `register`, and `logout` in the library's `routes/AuthRoutes.java`.

Things to notice about the code:

- `POST /logout` isn't a public route, so the before handler checks the token first. An invalid or missing token gets a `401` before the handler runs.
- The handler gets the token with `CurrentUser.bearerToken`, added in the previous slice.
- `ctx.status(204)` with no `ctx.json` sends a response with no body, as for `DELETE /photos/{id}`.

27 Wire It Together — `Main`

- Give `AuthRoutes` the `SessionService` too:

```
AuthRoutes authRoutes = new AuthRoutes(userService, sessionService);
```

28 Run and Test

```
# log out
curl -i -X POST -H "Authorization: Bearer $ARTHUR" \
     http://localhost:7000/logout
# the token no longer works
curl -i -H "Authorization: Bearer $ARTHUR" http://localhost:7000/photos
# logging out again
curl -i -X POST -H "Authorization: Bearer $ARTHUR" \
     http://localhost:7000/logout
# no token
curl -i -X POST http://localhost:7000/logout
```

Check that the session's row is gone from `SESSION`, then log in again to get a new token for further testing.

Administrators

The last two required endpoints are for administrators only. This slice doesn't add an endpoint, instead it sets up who the administrators are and the check that those endpoints will use.

Checking whether a user may do something has two parts: finding out which role the user has and then checking whether that role allows the action. The second part works the same way regardless of how the first is done. For the first part, there are three common ways to store the role information:

- **In the database.** Each user's role is stored with their account, as the library demo does with `MEMBERS.role`. The application can change roles while it runs (such as promoting a regular user to an administrator) and the change takes effect right away. This is the usual approach when the application manages its own users and it is what you should use in your project.
- **In an external identity provider.** This is what supports single sign-on systems — the application you log in to through SSO isn't checking your password itself, but instead uses a separate login system (an *identity provider*). The identity provider tells the application who you are; it can also tell what groups you belong to so the application can base roles on those instead of storing roles itself. This is common in organizations with many applications and one set of users.
- **In configuration.** The application reads the list of users who have a role from a configuration setting when it starts. This doesn't require any support for roles in the database, so it is simpler when a role applies only to a small, fixed set of users. It is less flexible, however, as roles can only be changed by editing the setting and restarting the application.

  A common use of configuration-based roles is to provide a way to bootstrap the first administrator of a new system — if roles are stored in the database and only administrators can assign roles, there's no way to create the first administrator within the application. Instead the first administrator can be named in a configuration setting and that user can then assign roles to others.

  Configuration-based roles also provide a way to support administrators without changing the database schema, and the photo album uses this strategy for that reason.

For the photo album, the administrators will be listed in an environment variable, `ADMIN_USERS`, set in your run configuration alongside `DB_URL` and the database credentials. Its value is a comma-separated list of usernames, such as `zaphod,trillian`. Any user not on the list is an ordinary user.

29 Create an Administrator

- Create an account to be the administrator:

```
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "zaphod", "password": "heartofgold"}' \
     http://localhost:7000/users
```

- In Eclipse, go to the Environment tab in the application's run configuration (where you set values for `DB_USER`, `DB_PASSWORD`, and `DB_URL`) and add the variable `ADMIN_USERS` with the value `zaphod`.

There's nothing special about zaphod's account in the database: what makes zaphod an administrator is being listed in `ADMIN_USERS`.

30 The Service — `UserService`

- `UserService` needs to know who the administrators are. Replace its instance variables and constructor with those shown in listing 17.
- Add the `isAdmin` and `requireAdmin` methods shown in listing 17.

Things to notice about the code:

- `requireAdmin` does the job of the library's `Auth.requireRole`, but in the service. The library demo can check in the route because its filter stores the whole `Member`, role included. Here, only `UserService` knows who the administrators are.
- It throws `ForbiddenException`, which `ErrorHandler` turns into a `403`: the user is logged in (or they'd have gotten a `401` already), but isn't allowed to do this.
- `requireAdmin` is for rules that apply to a whole endpoint, such as `GET /users` — only administrators can do that. `isAdmin` is for rules that combine being an administrator with something else, such as `DELETE /users/{username}` — regular users can delete their own account but only administrators can delete other accounts.

31 Wire It Together — `Main`

- Add the method shown in listing 18 to `Main`.
- In `main`, read the administrators before creating the services and pass them to `UserService`:

```
// Administrators: the usernames in ADMIN_USERS
Set<String> admins = adminUsers(System.getenv("ADMIN_USERS"));
System.out.println("Administrators: " + admins);
...
UserService userService = new UserService(userDao, sessionService, admins);
```

Things to notice about the code:

- The list is read once, when the server starts. To change who the administrators are, change `ADMIN_USERS` and restart the server. (Tokens keep working after a restart because sessions are stored in the database.)
- A `TreeSet` created with `String.CASE_INSENSITIVE_ORDER` compares usernames without regard to case, as MySQL does, so `Zaphod` in `ADMIN_USERS` would still match the stored `zaphod`.
- If `ADMIN_USERS` isn't set, there are no administrators. A missing setting means nobody gets extra permissions.
- The `println` is there so you can see which administrators the server found for debugging purposes; remove it once you have verified that things are working.

32 Run and Test

- Run `Main`. The console should show `Administrators: [zaphod]`. If it shows `Administrators: []`, check the variable in your run configuration.

The Remaining Slices

The last two endpoints are much like endpoints you wrote for homework #12, so the instructions for them are briefer and focus on the part that's new: who is allowed to use them.

33 `GET /users`

This retrieves a list of all of the photo album's users in alphabetical order.

This is much like `GET /photos`...

- add `findAll` to `UserDao`, similar to `PhotoDao.findAll` — order by username and retrieve all of the columns from `USER` even though not all of them will be passed back to the client
- add a `listUsers` method to `UserService`, similar to `PhotoService.listPhotos`
- add a `listUsers` route handler to `UserRoutes`, similar to `PhotoRoutes.listPhotos`
- register the route in `UserRoutes.register`, similar to how `GET /photos` was handled in `PhotoRoutes.register`

...with two modifications:

- The client needs to be sent `UserViews`, not `Users` — the client shouldn't get the password hashes! `GET /members` in the library has the same issue with `MemberView` and `Member`. See `MemberRoutes.list` and adapt for users.
- You'll notice that the first thing `MemberRoutes.list` does is check whether the current user is authorized for this task. It can because the library's filter stores the whole `Member`, role included, so the route already knows the current member's role. In the photo album, only `UserService` knows who the administrators are, so the check `requireAdmin(username);` will need to go as the first line of the service method instead.

Testing (log in as zaphod and save the token as `ZAPHOD` first):

```
# an administrator
curl -i -H "Authorization: Bearer $ZAPHOD" http://localhost:7000/users
# not an administrator (make sure arthur is logged in)
curl -i -H "Authorization: Bearer $ARTHUR" http://localhost:7000/users
# no token
curl -i http://localhost:7000/users
```

34 `DELETE /users/{username}`

This deletes a particular user account.

This is much like `DELETE /photos/{id}` — locate the DAO, service, and route methods that handle `DELETE /photos/{id}` and do a similar thing:

- add `delete` to `UserDao` — with a single username parameter (the user to delete) as there is no ownership check here
- add `deleteUser` to `UserService` — with *two* username parameters (the username to delete (comparable to the photo ID) and the current user)
- add `deleteUser` to `UserRoutes` — use `ctx.pathParam("username")` to get the username from the endpoint path (the username to delete) and `CurrentUser.get(ctx)` to get the current (logged in) user
- register the route in `UserRoutes.register`

The authorization check again goes into the service method. However, since the rule depends on which account is being deleted — users can delete their own account, administrators can delete any account — a simple `requireAdmin` alone doesn't work. The service method takes both the username from the path (the one to delete) and the current user's username, so it checks and throws `ForbiddenException` unless the usernames are the same or `isAdmin` says the current user is an administrator. Note:

- Compare the two usernames with `equalsIgnoreCase`, as MySQL would, so that a request to delete `/users/Trillian` made with trillian's token counts as a request to delete her own account.
- Check permission before existence. Someone who isn't allowed to delete an account should get a `403` whether or not it exists, so ordinary users can't use this endpoint to find out which usernames exist. Only an administrator gets a `404` for a username that doesn't exist (when the DAO's delete finds no row).
- The foreign keys do the rest: deleting a `USER` row deletes the user's photos (and so their keywords) and their sessions, logging the user out.

When testing, don't delete arthur or ford, whose photos the other tests rely on. Use accounts you create for testing purposes, such as the marvin account below.

Testing:

```
# an account to delete
curl -i -X POST -H "Content-Type: application/json" \
     -d '{"username": "marvin", "password": "paranoid"}' \
     http://localhost:7000/users
```

Log in as marvin and save the token (`MARVIN=...`). Then:

```
# marvin adds a photo to his album
curl -i -X POST -H "Authorization: Bearer $MARVIN" \
     -H "Content-Type: application/json" \
     -d '{"url": "http://bridgeman-vm.hws.edu/cs343f26/pics/IMG_4881.JPG"}' \
     http://localhost:7000/photos
# ... and it's there
curl -i -H "Authorization: Bearer $MARVIN" http://localhost:7000/photos
# arthur can't delete marvin's account (make sure arthur is logged in)
curl -i -X DELETE -H "Authorization: Bearer $ARTHUR" \
     http://localhost:7000/users/marvin
# ... and gets the same response for an account that doesn't exist
curl -i -X DELETE -H "Authorization: Bearer $ARTHUR" \
     http://localhost:7000/users/nobody
# marvin deletes his own account
curl -i -X DELETE -H "Authorization: Bearer $MARVIN" \
     http://localhost:7000/users/marvin
# marvin's token no longer works
curl -i -H "Authorization: Bearer $MARVIN" http://localhost:7000/photos
# an administrator deletes trillian's account
curl -i -X DELETE -H "Authorization: Bearer $ZAPHOD" \
     http://localhost:7000/users/trillian
# an administrator, an account that doesn't exist
curl -i -X DELETE -H "Authorization: Bearer $ZAPHOD" \
     http://localhost:7000/users/nobody
```

After deleting marvin's account, check in Workbench that marvin's photo and session were also deleted.

35 (Optional) `GET /users/me`

This should retrieve the current user and their role e.g. `{"username": "zaphod", "admin": true}`. It does the same thing as the library's `GET /members/me` except that whether or not the user is an admin has to be obtained from `UserService` instead of looking up the `Member` in the context.

Appendix

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

```java
package photoalbum.domain;

/**
 * One row of the USER table.
 *
 * Contains the password hash, so never send a User to the client; send a
 * UserView instead.
 */
public record User(String username, String passwordHash) {
}
```

Listing 1: `User.java`, for step 1 (The Domain Object — `User`)

```java
package photoalbum.dto;

/**
 * The body of POST /users: { "username": "...", "password": "..." }
 */
public record NewUser(String username, String password) {
}
```

Listing 2: `NewUser.java`, for step 2 (The DTOs — `NewUser` and `UserView`)

```java
package photoalbum.dto;

import photoalbum.domain.User;

/**
 * What the API sends back about a user: no password hash.
 */
public record UserView(String username) {

  public static UserView from(User user) {
    return new UserView(user.username());
  }
}
```

Listing 3: `UserView.java`, for step 2 (The DTOs — `NewUser` and `UserView`)

```
// The user with this username, or empty if there is none.
public Optional<User> findByUsername(String username) {
  String sql = "SELECT username, password FROM USER WHERE username = ?";
  try (Connection conn = DB.getConnection();
      PreparedStatement stmt = conn.prepareStatement(sql)) {
    stmt.setString(1, username);
    try (ResultSet rs = stmt.executeQuery()) {
      if (rs.next()) {
        return Optional.of(toUser(rs));
      }
      return Optional.empty();
    }
  } catch (SQLException e) {
    throw SqlErrors.translate(e);
  }
}

// Adds a user. The caller hashes the password; this method never sees
// the plaintext. A username that's already taken violates the primary
// key, which SqlErrors turns into a ConflictException.
public void insert(String username, String passwordHash) {
  String sql = "INSERT INTO USER (username, password) VALUES (?, ?)";
  try (Connection conn = DB.getConnection();
      PreparedStatement stmt = conn.prepareStatement(sql)) {
    stmt.setString(1, username);
    stmt.setString(2, passwordHash);
    stmt.executeUpdate();
  } catch (SQLException e) {
    throw SqlErrors.translate(e);
  }
}

// The current row of rs as a User.
private User toUser(ResultSet rs) throws SQLException {
  return new User(rs.getString("username"), rs.getString("password"));
}
```

Listing 4: `UserDao-addl`, for step 3 (The DAO — `UserDao`)

```
// USER.username is VARCHAR(50)
private static final int MAX_USERNAME_LENGTH = 50;
private static final int MIN_PASSWORD_LENGTH = 8;

// Creates an account and returns the new user.
// ValidationException for a bad username or password;
// ConflictException if the username is already taken.
public User register(NewUser newUser) {
  String username = newUser.username();
  if (username == null || username.isBlank()) {
    throw new ValidationException("username is required");
  }
  username = username.trim();   // an identifier: trim it
  if (username.length() > MAX_USERNAME_LENGTH) {
    throw new ValidationException("username must be at most "
        + MAX_USERNAME_LENGTH + " characters");
  }
  String password = newUser.password();   // never trimmed
  if (password == null || password.length() < MIN_PASSWORD_LENGTH) {
    throw new ValidationException("password must be at least "
        + MIN_PASSWORD_LENGTH + " characters");
  }

  // The plaintext exists only long enough to hash;
  // it's never stored or logged
  String hash = BCrypt.hashpw(password, BCrypt.gensalt());
  try {
    dao.insert(username, hash);
  } catch (ConflictException e) {
    // SqlErrors only knows "duplicate value"; say what was duplicated
    throw new ConflictException(
        "Username " + username + " is already taken", e);
  }

  Optional<User> created = dao.findByUsername(username);
  if (created.isEmpty()) {
    throw new AppException(
        "User " + username + " was inserted but can't be found");
  }
  return created.get();
}
```

Listing 5: `UserService-addl`, for step 4 (The Service — `UserService`)

```java
package photoalbum.routes;

import io.javalin.config.JavalinConfig;
import io.javalin.http.Context;
import photoalbum.domain.User;
import photoalbum.dto.NewUser;
import photoalbum.dto.UserView;
import photoalbum.service.UserService;

/**
 * The HTTP side of the /users endpoints.
 *
 * Responses use UserView, never User: a User carries the password hash.
 */
public class UserRoutes {

  private final UserService service;

  public UserRoutes(UserService service) {
    this.service = service;
  }

  // Called from inside Javalin.create(config -> { ... })
  public void register(JavalinConfig config) {
    config.routes.post("/users", ctx -> createUser(ctx));  // public
  }

  // POST /users -> 201 and the new user
  private void createUser(Context ctx) {
    NewUser body = ctx.bodyAsClass(NewUser.class);
    User created = service.register(body);
    ctx.status(201).json(UserView.from(created));
  }
}
```

Listing 6: `UserRoutes.java`, for step 5 (The Route — `UserRoutes`)

```
  // The requests that don't need a current user. Everything else does.
  private static boolean isPublic(Context ctx) {
    if (ctx.method().equals(HandlerType.POST)) {
      String path = ctx.path();
      // register, log in
      return path.equals("/users");
    }
    return false;
  }

// ---------------------------------------------------------------------

  // Called from inside Javalin.create(config -> { ... })
  public void register(JavalinConfig config) {
    config.routes.beforeMatched(ctx -> {
      if (isPublic(ctx)) {
        return;  // no current user needed
      }
      String username = ctx.header("X-User");
      if (username == null || username.isBlank()) {
        throw new UnauthorizedException("X-User header is required");
      }
      username = username.trim();
      service.requireUser(username);
      ctx.attribute(KEY, username);
    });
  }
```

Listing 7: `CurrentUser-addl`, for step 6 (Make It Public — `CurrentUser`)

```java
package photoalbum.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.time.LocalDateTime;

import photoalbum.db.DB;

/**
 * All of the SQL for the SESSION table.
 */
public class SessionDao {

  // Stores a new session: a token for this user, valid until expiresAt.
  public void insert(String token, String username,
      LocalDateTime expiresAt) {
    String sql = "INSERT INTO SESSION (token, username, expires_at) "
        + "VALUES (?, ?, ?)";
    try (Connection conn = DB.getConnection();
        PreparedStatement stmt = conn.prepareStatement(sql)) {
      stmt.setString(1, token);
      stmt.setString(2, username);
      stmt.setObject(3, expiresAt);
      stmt.executeUpdate();
    } catch (SQLException e) {
      throw SqlErrors.translate(e);
    }
  }
}
```

Listing 8: `SessionDao.java`, for step 11 (The DAO — `SessionDao`)

```java
package photoalbum.service;

import java.security.SecureRandom;
import java.time.Duration;
import java.time.LocalDateTime;
import java.util.Base64;

import photoalbum.dao.SessionDao;
import photoalbum.domain.User;

/**
 * Login sessions, stored in the SESSION table.
 */
public class SessionService {

  // SecureRandom, not Random: tokens must be unguessable
  private static final SecureRandom RANDOM = new SecureRandom();
  private static final Duration SESSION_LENGTH = Duration.ofHours(8);

  private final SessionDao dao;

  public SessionService(SessionDao dao) {
    this.dao = dao;
  }

  // Starts a session for this user and returns its token.
  public String start(User user) {
    byte[] bytes = new byte[32];
    RANDOM.nextBytes(bytes);
    String token = Base64.getUrlEncoder().withoutPadding()
        .encodeToString(bytes);
    dao.insert(token, user.username(),
        LocalDateTime.now().plus(SESSION_LENGTH));
    return token;
  }
}
```

Listing 9: `SessionService.java`, for step 12 (The Session Service — `SessionService`)

```
private static final String INVALID_LOGIN =
    "Invalid username or password";

private final UserDao dao;
private final SessionService sessionService;

public UserService(UserDao dao, SessionService sessionService) {
  this.dao = dao;
  this.sessionService = sessionService;
}

// ------------------------------------------------------------------------

// Logs in and returns a new session token.
// UnauthorizedException for an unknown username or a wrong password,
// with the same message for both, so the response doesn't reveal which
// usernames exist.
public String login(String username, String password) {
  if (username == null || password == null) {
    throw new UnauthorizedException(INVALID_LOGIN);
  }
  Optional<User> user = dao.findByUsername(username.trim());
  if (user.isEmpty()
      || !BCrypt.checkpw(password, user.get().passwordHash())) {
    throw new UnauthorizedException(INVALID_LOGIN);
  }
  // the stored username, which may differ in case from what was typed
  return sessionService.start(user.get());
}
```

Listing 10: `UserService-addl2`, for step 13 (The Service — `UserService`)

```java
package photoalbum.routes;

import io.javalin.config.JavalinConfig;
import io.javalin.http.Context;
import photoalbum.dto.LoginRequest;
import photoalbum.dto.TokenResponse;
import photoalbum.service.UserService;

/**
 * The HTTP side of logging in (and, later, out).
 */
public class AuthRoutes {

  private final UserService userService;

  public AuthRoutes(UserService userService) {
    this.userService = userService;
  }

  // Called from inside Javalin.create(config -> { ... })
  public void register(JavalinConfig config) {
    config.routes.post("/login", ctx -> login(ctx));  // public
  }

  // POST /login -> 200 and a session token
  private void login(Context ctx) {
    LoginRequest body = ctx.bodyAsClass(LoginRequest.class);
    String token = userService.login(body.username(), body.password());
    ctx.json(new TokenResponse(token));
  }
}
```

Listing 11: `AuthRoutes.java`, for step 14 (The Route — `AuthRoutes`)

```
// The username a token belongs to, if the session exists and hasn't
// expired. "Now" comes from Java, as expires_at did, so both use the
// same clock.
public Optional<String> findUsername(String token) {
  String sql = "SELECT username FROM SESSION "
      + "WHERE token = ? AND expires_at > ?";
  try (Connection conn = DB.getConnection();
      PreparedStatement stmt = conn.prepareStatement(sql)) {
    stmt.setString(1, token);
    stmt.setObject(2, LocalDateTime.now());
    try (ResultSet rs = stmt.executeQuery()) {
      if (rs.next()) {
        return Optional.of(rs.getString("username"));
      }
      return Optional.empty();
    }
  } catch (SQLException e) {
    throw SqlErrors.translate(e);
  }
}

// ---------------------------------------------------------------------

// Ends a session. Returns false if there was no such token.
public boolean delete(String token) {
  String sql = "DELETE FROM SESSION WHERE token = ?";
  try (Connection conn = DB.getConnection();
      PreparedStatement stmt = conn.prepareStatement(sql)) {
    stmt.setString(1, token);
    return stmt.executeUpdate() == 1;
  } catch (SQLException e) {
    throw SqlErrors.translate(e);
  }
}
```

Listing 12: `SessionDao-addl`, for steps 18 (The DAO — `SessionDao`) and 24 (The DAO — `SessionDao`)

```java
package photoalbum.routes;

import java.util.Optional;

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

/**
 * Works out whose album a request is for: the user who the request's login
 * token belongs to. Routes get the current user with CurrentUser.get(ctx).
 */
public class CurrentUser {

  private static final String KEY = "currentUser";
  private static final String BEARER = "Bearer ";

  private final SessionService sessionService;

  public CurrentUser(SessionService sessionService) {
    this.sessionService = sessionService;
  }

  // Called from inside Javalin.create(config -> { ... })
  public void register(JavalinConfig config) {
    config.routes.beforeMatched(ctx -> {
      if (isPublic(ctx)) {
        return;  // no current user needed
      }
      Optional<String> token = bearerToken(ctx);
      if (token.isEmpty()) {
        throw new UnauthorizedException("Login required");
      }
      Optional<String> username =
          sessionService.usernameForToken(token.get());
      if (username.isEmpty()) {
        // unknown or expired token
        throw new UnauthorizedException("Login required");
      }
      ctx.attribute(KEY, username.get());
    });
  }
```

Listing 13: `CurrentUser.java-v2`, for step 20 (Check the Token — `CurrentUser`)

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

  // The token from "Authorization: Bearer <token>", if the header is there.
  public static Optional<String> bearerToken(Context ctx) {
    String header = ctx.header("Authorization");
    if (header == null || !header.startsWith(BEARER)) {
      return Optional.empty();
    }
    return Optional.of(header.substring(BEARER.length()));
  }

  // The requests that don't need a current user. Everything else does.
  private static boolean isPublic(Context ctx) {
    if (ctx.method().equals(HandlerType.POST)) {
      String path = ctx.path();
      // register, log in
      return path.equals("/users") || path.equals("/login");
    }
    return false;
  }
}
```

Listing 14: `CurrentUser.java-v2` (continued), for step 20 (Check the Token — `CurrentUser`)

```
// Ends the session with this token (logging out).
public void end(String token) {
  dao.delete(token);
}
```

Listing 15: `SessionService-addl`, for step 25 (The Session Service — `SessionService`)

```java
package photoalbum.routes;

import java.util.Optional;

import io.javalin.config.JavalinConfig;
import io.javalin.http.Context;
import photoalbum.dto.LoginRequest;
import photoalbum.dto.TokenResponse;
import photoalbum.service.SessionService;
import photoalbum.service.UserService;

/**
 * The HTTP side of logging in and out.
 */
public class AuthRoutes {

  private final UserService userService;
  private final SessionService sessionService;

  public AuthRoutes(UserService userService,
      SessionService sessionService) {
    this.userService = userService;
    this.sessionService = sessionService;
  }

  // Called from inside Javalin.create(config -> { ... })
  public void register(JavalinConfig config) {
    config.routes.post("/login", ctx -> login(ctx));  // public
    config.routes.post("/logout", ctx -> logout(ctx));  // needs a token
  }

  // POST /login -> 200 and a session token
  private void login(Context ctx) {
    LoginRequest body = ctx.bodyAsClass(LoginRequest.class);
    String token = userService.login(body.username(), body.password());
    ctx.json(new TokenResponse(token));
  }

  // POST /logout -> 204. The before handler has already checked the
  // token; this ends its session.
  private void logout(Context ctx) {
    Optional<String> token = CurrentUser.bearerToken(ctx);
    if (token.isPresent()) {
      sessionService.end(token.get());
    }
    ctx.status(204);
  }
}
```

Listing 16: `AuthRoutes.java-v2`, for step 26 (The Route — `AuthRoutes`)

```
private final UserDao dao;
private final SessionService sessionService;
private final Set<String> admins;  // usernames of the administrators

public UserService(UserDao dao, SessionService sessionService,
    Set<String> admins) {
  this.dao = dao;
  this.sessionService = sessionService;
  this.admins = admins;
}

// ---------------------------------------------------------------------

// Whether this user is an administrator.
public boolean isAdmin(String username) {
  return admins.contains(username);
}

// ForbiddenException unless this user is an administrator.
public void requireAdmin(String username) {
  if (!isAdmin(username)) {
    throw new ForbiddenException("Administrators only");
  }
}
```

Listing 17: `UserService-addl3`, for step 30 (The Service — `UserService`)

```
// The usernames in a comma-separated list such as "zaphod, trillian".
// Compared without regard to case, as MySQL compares usernames.
// Empty if the list is null (the environment variable isn't set).
private static Set<String> adminUsers(String list) {
  Set<String> admins = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
  if (list != null) {
    for (String name : list.split(",")) {
      if (!name.isBlank()) {
        admins.add(name.trim());
      }
    }
  }
  return admins;
}
```

Listing 18: `Main-addl`, for step 31 (Wire It Together — `Main`)