Authentication and Authorization

- authentication: *who are you?*
- authorization: *what may you do?*

- these are application-level users, not MySQL users
  - user information is stored in a table
  - the application is responsible for who can do what
- the application connects to MySQL using a single database account

Storing Credentials

- never store the password directly
  - assume info in the database will leak someday – stolen backup, injection bug, misconfigured server, …
- encrypted passwords aren't much better
  - encryption is designed to be reversible
    - in symmetric key encryption, the same key encrypts and decrypts
  - application needs to use (and thus store) the encryption key
    - checks passwords by encrypting the user's attempt and comparing to the stored version
- best practice is a *slow, salted hash*
  - hashing is one-way – cannot recover the original from the hash
    - only hashed password is stored; to check a login, hash the attempt and compare
  - no key to store, but fast hashes (MD5, SHA-1, plain SHA-256) are vulnerable to dictionary attacks
  - a slow hash (bcrypt, scrypt, Argon2id) means each guess is expensive, making dictionary attacks less effective
  - *salt* is random bytes included with each password
    - salt is not secret, but it ensures that the same password hashes differently in different instances, thwarting rainbow tables

Login and Tokens

- log in once, send a token every time
  - REST is stateless – the server remembers nothing between requests so the client must prove who it is on every request
- why use a token instead of the password
  - less exposure for the password – if the client has to send it every time, the client has to store it
  - requests stay cheap – bcrypt is deliberately slow while a token check is just a lookup
  - stolen tokens do less damage – the token only works with this API, and it expires
    - (users often reuse passwords for different applications)
  - potential for access to be revoked – deleting a stored token logs the client out

Managing Tokens

- session-based authentication
  - on successful login, the server generates a unique session ID
    - server stores active session IDs
  - client sends token with subsequent API requests
    - server verifies token via lookup
  - server controls session data – can update permissions dynamically and revoke tokens prior to expiration (e.g. on logout)
    - to revoke, server deletes from stored session IDs
  - simple but doesn't scale well – resource intensive, increased complexity in distributed systems
- JWT (JSON Web Token)
  - server signs a payload (identity of user, token expiry), then verifies signature on usage
  - stateless and scalable – server doesn't store anything and token can be verified without lookup
  - cannot revoke token before it expires or force logout of an active session
    - can manage with a combination of short-lived access tokens and longer-lived refresh tokens allowing users to obtain new access tokens

Managing Tokens

- in production systems, must use HTTPS to protect the transmission of the tokens
  - more critical with JWT – a stolen token can be used until it expires regardless of whether the user has logged out or the server has detected a stolen token
- detecting stolen tokens
  - with session tokens, the server can store timestamp, IP address, browser of last usage
    - if a suspicious usage is seen, revoke token and force a new login
  - JWT uses refresh-token rotation – a new refresh token is issued each time one is used
    - if an old refresh token is used, force a new login

Implementation

- updates to the DB tables

table	columns	notes
`MEMBERS`	`username` (`UNIQUE`), `password_hash`, `role`	store the username, the bcrypt string (hashed password + salt), role (member or librarian)
`SESSIONS` (new)	`token` (PK), `member_id` (FK), `expires_at`	one row per logged-in session

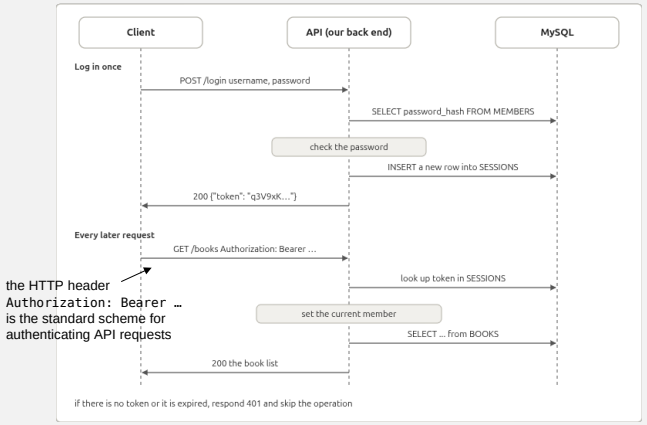
- new endpoints

use case	endpoint	service method	token required?
register	`POST /members`	`MemberService.register`	no
logged in as?	`GET /members/me`	returns the already-looked up member from the session	yes
log in	`POST /login`	`MemberServer.login`	no
log out	`POST /logout`	`SessionService.end`	yes
all others	its own endpoint, plus `Authorization: Bearer …`	`AuthFilter` checks the token first, then its own service method	yes

Implementation

- new classes
  - `SessionDao` – all SQL for the `SESSIONS` table
    - no corresponding `Session` domain object because nothing in the application reads a session row back as an object – sessions are implementation-only
  - `MemberService` – manage member-related operations (register, login)
  - `SessionService` – manage token-related operations (token generation, lookup, logout)
  - `MemberRoutes`, `AuthRoutes` – manage the HTTP in and out for the new endpoints
    - `AuthRoutes` instead of `SessionRoutes`: name route classes after the resource when there is one and after their purpose otherwise
      - `SessionRoutes` would be appropriate if the endpoints were `POST /sessions`, `DELETE /sessions/current`
  - `AuthFilter` – checks the token
- new handling
  - check the token before every route

Login Token Flow



the HTTP header `Authorization: Bearer …` is the standard scheme for authenticating API requests

Register a New User

- validate the request, hash the password, insert a new member

```
public Member register(NewMember nm) throws SQLException {
    if (nm.username() == null || nm.username().isBlank()) {
        throw new IllegalArgumentException("username is required");
    }
    if (nm.password() == null || nm.password().isEmpty()) {
        throw new IllegalArgumentException("password is required");
    }
    String hash = BCrypt.hashpw(nm.password(), BCrypt.gensalt());
    int id = memberDao.insert(nm.username(), hash, "member");
    Optional<Member> created = memberDao.findById(id);
    if (created.isEmpty()) {
        throw new IllegalStateException("Member " + id +
                                        " was inserted but can't " +
                                        " be found");
    }
    return created.get();
}
```

`isBlank` checks for whitespace-only strings as well as length 0 (empty) strings

whitespace-only usernames would cause confusion

whitespace-only passwords aren't ideal but spaces are legitimate characters, and current guidance is to accept every printable character and to never trim or alter a password

the role is set by the server – all new members are `member`, not `librarian`

- this shouldn't happen, and if it does, it isn't the client's fault – nothing the client can do
- `IllegalStateException` is the standard Java exception for "this shouldn't happen" and will get handled by the default handler which produces a generic 500 (server's fault) error

Register a New User

```
public Member register(NewMember nm) throws SQLException {
    if (nm.username() == null || nm.username().isBlank()) {
        throw new IllegalArgumentException("username is required");
    }
    if (nm.password() == null || nm.password().isEmpty()) {
        throw new IllegalArgumentException("password is required");
    }
    String hash = BCrypt.hashpw(nm.password(), BCrypt.gensalt());
    int id = memberDao.insert(nm.username(), hash, "member");
    Optional<Member> created = memberDao.findById(id);
    if (created.isEmpty()) {
        throw new IllegalStateException("Member " + id +
                                        " was inserted but can't " +
                                        " be found");
    }
    return created.get();
}
```

- `NewMember` captures just the information sent by the client

```
public record NewMember(String username, String password) {}
```

- uses the jBCrypt library – bcrypt is a slow, salted hash with adjustable cost
  - the conventional choice – the original Java port of OpenBSD's bcrypt
  - a safe choice – trusted author, small enough to read (so many people have audited it), known bugs fixed
  - for production code use the stronger Argon2id or scrypt instead (memory intensive)

Login

- fetch the user's hash, check the attempt
  - if successful, assign a token

```
public Optional<String> login(String username, String password)
 throws SQLException {
    Optional<Member> member = memberDao.findByUsername(username);
    if (member.isEmpty() ||
        !BCrypt.checkpw(password, member.get().passwordHash())) {
        return Optional.empty();
    }
    return Optional.of(sessionService.start(member.get()));
}
```


Managing Session Tokens

- tokens must be random and unguessable, expire, and be removed on logout
  - `Random` is fast but predictable – never use it for sensitive applications!
  - `SecureRandom` is cryptographically secure (but more expensive)

```
// SessionService
private static final SecureRandom RANDOM = new SecureRandom();
private static final Duration SESSION_LENGTH = Duration.ofHours(8);

public String start(Member m) throws SQLException {
    byte[] bytes = new byte[32];
    RANDOM.nextBytes(bytes);
    String token = Base64.getUrlEncoder().withoutPadding().encodeToString(bytes);
    sessionDao.insert(token, m.memberId(),
                      LocalDateTime.now().plus(SESSION_LENGTH));
    return token;
}

public Optional<Member> memberForToken(String token) throws SQLException {
    return sessionDao.findMember(token);   // find the member whose token this is
}

public void end(String token) throws SQLException {
    sessionDao.delete(token);              // logout
}
```

`LocalDateTime`, `Duration` are part of the modern Java Date and Time API

convert the raw bytes to printable characters – JSON, HTTP headers, and VARCHAR all need text

Checking Tokens

- must verify the token for (nearly) every route
  - implement as a *before filter* – makes the default locking the route (require checking a token) rather than public (open)

Checking Tokens

```
public class AuthFilter implements Handler {
    private final SessionService sessions;

    public AuthFilter(SessionService sessions) {
        this.sessions = sessions;
    }

    @Override
    public void handle(Context ctx) throws Exception {
        if (isPublic(ctx)) {
            return;
        }
        Optional<String> token = Auth.bearerToken(ctx);
        if (token.isEmpty()) {              // no Authorization: Bearer header
            throw new UnauthorizedException("Missing token");
        }
        Optional<Member> member = sessions.memberForToken(token.get());
        if (member.isEmpty()) {             // unknown or expired token
            throw new UnauthorizedException("Invalid or expired token");
        }
        ctx.attribute(Auth.MEMBER, member.get());
    }

    private boolean isPublic(Context ctx) {
        return ctx.method().equals(HandlerType.POST)
            && (ctx.path().equals("/login") || ctx.path().equals("/members"));
    }
}
```

attach the authenticated `Member` to this request's context so the route can get it without another lookup

only login and registering a new member are public – no token required

Checking Tokens – Helpers

```
public final class Auth {

    public static final String MEMBER = "member";
    private static final String BEARER = "Bearer ";

    private Auth() {}

    // 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 logged-in member. AuthFilter has already run, so on a protected route
    // this is never null.
    public static Member currentMember(Context ctx) {
        return ctx.attribute(MEMBER);
    }
}
```

the HTTP header `Authorization: Bearer …` is the standard scheme for authenticating API requests

`substring(…)` strips off the "Bearer " part of the header value

Configuring Routes with a Filter

```
// in main
Javalin.create(config -> {
    config.jetty.port = 7000;

    config.routes.before(new AuthFilter(sessionService)); // runs before every route

    authRoutes.register(config);
    bookRoutes.register(config);
    memberRoutes.register(config);
}).start();
```

Roles

- some endpoints should be librarian-only

add a new book	`POST /books`
edit a book's details	`PUT /books/{id}`
remove a book	`DELETE /books/{id}`

Implementing Roles

- add a helper to `Auth` for checking that the logged-in member has a role

```
public final class Auth {
    …
    static Member requireRole(Context ctx, String role) {
        Member current = ctx.attribute("member");
        if (!current.role().equals(role)) {    // logged in but not allowed
            throw new ForbiddenException("Requires the " + role + " role");
        }
        return current;
    }
}
```

- check at the beginning of role-restricted routes

```
public class BookRoutes {
    …
    private void create(Context ctx) throws SQLException {
        Auth.requireRole(ctx, Roles.LIBRARIAN);
        // …the rest of the route, as before…
    }
    …
}
```

Row-Level Authorization

- a role check is sufficient for rules about whole endpoints
  - e.g. members can never add books
- row-level authorization is needed when users can only access certain rows

use case	endpoint	member	librarian
list loans	`GET /loans`	their own loans	every loan
view one loan	`GET /loans/{id}`	only if it is theirs, otherwise 404	any loan
check out a book	`POST /loans`	for themselves	for themselves
return a book	`PUT /loans/{id}/return`	only if it is theirs, otherwise 404	any loan

Implementation

- ownership is already in the data
  - `LOANS.member_id` records whose loan this is
- the current member comes from the session, never from the request
  - the session stores the authenticated member
- the service picks the query based on the role
- the DAO puts the owner in the `WHERE`
  - every read, update, and delete of an owned row needs the check

Implementation – Routes

```
// LoanRoutes

private void list(Context ctx) {                        // GET /loans
    ctx.json(service.listLoans(Auth.currentMember(ctx)));
}

private void getOne(Context ctx) {                      // GET /loans/{id}
    ctx.json(service.getLoan(Auth.currentMember(ctx), pathId(ctx)));
}
```

member id comes from the session, not the request

Implementation – Services

```
// LoanService

public Loan getLoan(Member current, int loanId) {
    Optional<Loan> loan;
    if (current.role().equals("librarian")) {       // any loan
        loan = loanDao.findById(loanId);
    } else {                                        // own loans only
        loan = loanDao.findByIdForMember(loanId, current.memberId()); only
    }
    if (loan.isEmpty()) {                          // missing or not yours
        throw new NotFoundException("No loan with id " + loanId);
    }
    return loan.get();
}

public List<Loan> listLoans(Member current) {
    if (current.role().equals("librarian")) {
        return loanDao.findAll();
    }
    return loanDao.findByMember(current.memberId());
}
```

- the service picks the query – librarians see every loan, members see only their own

Implementation – DAO

```
// LoanDao

private static final String COLUMNS =
        "loan_id, book_id, member_id, checked_out_on, due_on, returned_on";

public Optional<Loan> findByIdForMember(int loanId, int memberId) {
    String sql = "SELECT " + COLUMNS + " FROM LOANS WHERE loan_id = ? AND member_id = ?";
    try (Connection conn = Db.getConnection();
         PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(1, loanId);
        ps.setInt(2, memberId);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) { return Optional.of(map(rs)); }
            return Optional.empty();
        }
    } catch (SQLException e) { throw SqlErrors.translate(e); }
}

public List<Loan> findByMember(int memberId) {
    String sql = "SELECT " + COLUMNS + " FROM LOANS WHERE member_id = ? "
               + "ORDER BY checked_out_on DESC";
    List<Loan> loans = new ArrayList<>();
    try (Connection conn = Db.getConnection();
         PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(1, memberId);
        try (ResultSet rs = ps.executeQuery()) {
            while (rs.next()) { loans.add(map(rs)); }
        }
    } catch (SQLException e) { throw SqlErrors.translate(e); }
    return loans;
}
```

the DAO puts the owner in the WHERE

IDOR

- *insecure direct object references* (IDOR) are a type of access control vulnerability
  - occur when an application uses user-supplied input to access objects directly
- scenario – `GET /loans/{id}`
  - `{id}` is a direct object reference (to a specific row of the `LOANS` table) supplied by the user (insecure)
    - can't trust that the user is only asking for their own loans!
  - must include a check that only retrieves the loan if it belongs to the logged-in user

```
-- Vulnerable: any logged-in member can read any loan
SELECT loan_id, book_id, member_id, checked_out_on, due_on, returned_on
FROM LOANS
WHERE loan_id = ?

-- Fixed: the row must also belong to the current member
SELECT loan_id, book_id, member_id, checked_out_on, due_on, returned_on
FROM LOANS
WHERE loan_id = ? AND member_id = ?
```

Status Codes and Leaking Information

- status code and error messages are part of what an API reveals – choose them deliberately

situation	code	meaning
no token, bad token, expired token	401 Unauthorized	we don't know who you are
member tries `POST /books`	403 Forbidden	we know who you are; the answer is no
member asks for someone else's loan	404 Not Found	as far as you're concerned, it doesn't exist
wrong username or wrong password	401, one message for both cases ("invalid username or password")	invalid username or password

  - 401, 403 are standard responses for authentication and whole-endpoint lack-of-authorization
  - 404 instead of 403 for someone else's info – 403 confirms that the requested loan exists, allowing an attacker to determine which ids are real
  - separate "no such user" and "wrong password" messages tells an attacker which usernames exist

Avoiding Leakage

- production systems employ additional measures
- timing
  - response time for wrong username or password also leaks info – no bcrypt check for an unknown user

    → run a dummy check when the user is unknown to equalize response times
- registration
  - `409 Conflict` response for `POST /members` with a duplicate username reveals that the username exists

    → sites with public usernames accept the leak – those names are visible anyway

    → sites with private usernames can use signup by email – respond to every registration with "check your email"
    - send a link to complete registration only if the address is new
- rate limiting and lockout
  - limit how many login attempts an IP address or account can make to slow down password guessing and exploiting leaks to find usernames

Avoiding Leakage

- status code and error messages are part of what an API reveals – choose them deliberately

situation	code	meaning
no token, bad token, expired token	401 Unauthorized	we don't know who you are
member tries `POST /books`	403 Forbidden	we know who you are; the answer is no
member asks for someone else's loan	404 Not Found	as far as you're concerned, it doesn't exist
wrong username or wrong password	401, one message for both cases ("invalid username or password")	invalid username or password

  - 401, 403 are standard responses for authentication and whole-endpoint lack-of-authorization
  - 403 for someone else's info confirms that the requested loan exists, allowing an attacker to determine which ids are real
  - separate "no such user" and "wrong password" messages tells an attacker which usernames exist

Putting It All Together – Authentication and Authorization

element	does	can fail with
`AuthFilter`	looks up the token from the `Authorization` header and puts the current member on the context (public routes skip it)	401
Route	parses the path id, query parameters, and JSON body; checks the role for librarian-only endpoints; calls one service method	400 (bad id or body), 403
	sets the success status, any headers (such as `Location` after a create), and the JSON body	on success: 200, 201, 204
Service	enforces the library's rules (a book must exist and be available), limits members to their own rows, and runs the steps of a use case in order, across DAOs	400, 404, 409
DAO	runs the SQL through JDBC, maps rows to domain objects, and translates any `SQLException` into one of our exceptions	409, 404, 400, 503
MySQL	stores the data and enforces the constraints (`UNIQUE`, foreign keys, `NOT NULL`): the last line of defense if a check above was missed	(reported through the DAO)
`ErrorHandler`	catches any exception thrown above, chooses the status code, logs the details of 5xx failures, and sends the error JSON	500 for anything unexpected