Users

There are two kinds of users –

- *application users* managed by the application
  - user information is stored in a table in the database (e.g. `MEMBERS`)
  - the application/API authenticates them
- *database accounts* managed by MySQL
  - MySQL authenticates them when a connection is established
- the application/API connects with one database account (`DB_USER` in `DB.java`) for every request
  - MySQL doesn't know which application user is asking, only which database account is connected
- determining who can do what (authorization)
  - the application/API determines and enforces for application users
  - MySQL determines and enforces for database accounts
  - the application's database account limits what every application user can do, even if the application has a bug

Accounts and Privileges

- a database account is a user name plus the host it may connect from
  - the same username from a different host is a different account
  - '%' allows connections from anywhere
- a new account can do nothing until granted privileges
- `GRANT` adds privileges, `REVOKE` removes them, `SHOW GRANTS` lists them

```
CREATE USER 'library_app'@'localhost' IDENTIFIED BY '...';

GRANT SELECT, INSERT, UPDATE, DELETE ON demo_library.BOOKS TO
'library_app'@'localhost';
GRANT SELECT, INSERT, UPDATE ON demo_library.LOANS TO 'library_app'@'localhost';

REVOKE DELETE ON demo_library.BOOKS FROM 'library_app'@'localhost';

SHOW GRANTS FOR 'library_app'@'localhost';
```


Privileges

- many privileges correspond to kinds of SQL statements
  - (data) SELECT, INSERT, UPDATE, DELETE
  - (schema) CREATE, ALTER, DROP, INDEX, CREATE VIEW, TRIGGER, …
  - (stored programs) EXECUTE, CREATE ROUTINE, …
  - statements can need more that their own privilege
    - e.g. an UPDATE or DELETE whose WHERE clause reads columns also needs SELECT
- others control capabilities
  - REFERENCES – create foreign keys that point to a table
  - GRANT OPTION – pass your privileges on to others
  - administrative – CREATE USER, PROCESS, FILE, SHUTDOWN, …
- not an exhaustive list!

How Fine-Grained?

- privileges can apply at different levels

level	example	means
global	`ON *.*`	every database on the server
database	`ON demo_library.*`	every table in one database
table	`ON demo_library.BOOKS`	one table
column	`SELECT (member_id,username,role) ON demo_library.MEMBERS`	only those columns

  - column privileges can be tricky
    - e.g. `SELECT * FROM MEMBERS` would fail in this case because `*` includes additional columns (`password_hash`)
  - views are often a better way to hide columns
- many administrative privileges are global

Least Privilege

- an account should have only what it needs and no more
  - the *least privilege* possible
- an application should never connect as `root`
  - most administrative privileges are global

account	used by	privileges
library_app	the API	SELECT, INSERT, UPDATE, DELETE on BOOKS SELECT, INSERT, UPDATE on LOANS SELECT, INSERT on MEMBERS SELECT, INSERT, DELETE on SESSIONS
library_report	a reporting tool	SELECT on two views only
library_admin	you, setting up the library database	everything on demo_library, including CREATE and DROP

  - the library API never changes the schema, so no CREATE, ALTER, DROP, or GRANT OPTION
  - no DELETE on LOANS because the API doesn't include that operation (returning a book leaves the loan as a history)

Denied

```
mysql -u library_report -p demo_library

mysql> DELETE FROM BOOKS WHERE book_id = 42;
ERROR 1142 (42000): DELETE command denied to user 'library_report'@'localhost'
for table 'BOOKS'
```

- SqlErrors turns error 1142 into a generic HTTP 500 Internal Server Error
  - code and grants disagree – a bug to fix, not something the API's client did wrong

Roles

- a *role* is a named bundle of privileges
  - grant privileges to the role once, then grant the role to each account that needs it
  - change the role and every account's privileges change too

```
CREATE ROLE 'reporting';
GRANT SELECT ON demo_library.CURRENT_LOANS TO 'reporting';
GRANT 'reporting' TO 'library_report'@'localhost';
SET DEFAULT ROLE ALL TO 'library_report'@'localhost';
```

  - GRANT only says what roles an account is allowed to take on
    - it doesn't cause those roles to be assigned to the account on login – a MySQL role is inactive until it has been activated
  - SET DEFAULT ROLE assigns roles at login
    - ALL assigns all the roles granted
    - specific roles can be listed to assign only those roles
      - roles to be assigned must have been granted to the account

Views

- a *view* is a named, stored query that can be used like a table
  - can use SELECT, WHERE, joins
  - some views can be updated, though views are mostly used for reading
- it stores the query, not the data – every use runs the query against the current tables
- define with `CREATE VIEW name AS SELECT …`
- remove with `DROP VIEW name`

```
CREATE VIEW LOAN_DETAILS AS
  SELECT L.loan_id, L.member_id, B.title, L.due_on, L.returned_on
  FROM LOANS L JOIN BOOKS B ON B.book_id = L.book_id;

SELECT title, due_on FROM LOAN_DETAILS
WHERE member_id = 7 AND returned_on IS NULL;
```

Views As Security Mechanism

- privileges can be granted to views
  - does not confer privileges on the base tables – the account can see only what is in the view, not where the data came from
- when a view runs, the base tables are read with the privileges of the view's creator (its *definer*), not the account querying it
  - the view definer needs access to the base tables
  - the reader needs access only to the view
- privileges decide which objects an account can reach
- views decide what those objects show

Views As Security Mechanism

- limiting columns

```
CREATE VIEW MEMBER_DIRECTORY AS
  SELECT member_id, username, role FROM MEMBERS;
```

  - the view excludes the `password_hash` column
    - unreachable for accounts which only have access to the view
  - view must be defined by an account with access to MEMBERS
- limiting rows

```
CREATE VIEW CURRENT_LOANS AS
  SELECT loan_id, book_id, member_id, due_on
  FROM LOANS
  WHERE returned_on IS NULL;
```

  - the WHERE clause limits the rows visible in the view
    - can only see loans still out, not a member's borrowing history
  - views can't filter by application user – MySQL doesn't know about application users
    - must be done by the application

SQL Injection

```
// GET /books?author=...  (deliberately vulnerable version)
String sql = "SELECT " + COLUMNS +
             " FROM BOOKS WHERE author = '" + author + "' ORDER BY title";
try (Connection conn = DB.getConnection();
     Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(sql)) {
    ...
}
```

- what happens for `?author=Adams`?

```
SELECT ... FROM BOOKS WHERE author = 'Adams' ORDER BY title
```

- what happens for `?author=O'Brien`?

```
SELECT ... FROM BOOKS WHERE author = 'O'Brien' ORDER BY title
```

- what happens for `?author=x' OR '1'='1`?

```
SELECT ... FROM BOOKS WHERE author = 'x' OR '1'='1' ORDER BY title
```

- what happens for `?author=x' UNION SELECT member_id, username, password_hash, role, NULL, NULL, TRUE FROM MEMBERS --`?

```
SELECT ... FROM BOOKS WHERE author = 'x' UNION SELECT member_id, username,
password_hash, role, NULL, NULL, TRUE FROM MEMBERS --' ORDER BY title
```


Stopping Injection

```
String sql = "SELECT " + COLUMNS +
             " FROM BOOKS WHERE author = ? ORDER BY title";
try (Connection conn = DB.getConnection();
     PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, author);
    ...
}
```

- with placeholders, the query's structure is fixed by the code – input can only ever be a value
- values are either sent separately and never parsed or are escaped by the driver into quoted literals (depends on the driver)
  - both are safe
    - not going through the SQL parser means that the value can't ever be interpreted as part of the query structure
    - the driver knows the connection's character set and all the quoting rules and it applies them in every case – hand-escaping is prone to mistakes

What Placeholders Can't Do

- ? stands for a value (only), not for names (table or column) or other elements of a query (e.g. `ASC`/`DESC`)
- when those values come from a user, map the input to a fixed *allow-list* of names

```
// GET /books?sort=... : only these names ever reach the SQL
private static String sortColumn(String sort) {
    if (sort == null || sort.equals("title")) {
        return "title";
    }
    if (sort.equals("author")) {
        return "author";
    }
    if (sort.equals("published")) {
        return "published_on";
    }
    throw new ValidationException("sort must be title, author, or published");
}
```

  - "… `ORDER BY ` " + `sortColumn(sort)` is safe because what is concatenated is only ever one of three values

What Placeholders Can't Do

- gotcha alert: `ORDER BY ?` doesn't fail (no error) but it also doesn't work
  - ? is replaced by a value, so using `title` results in `ORDER BY 'title'`



- `ORDER BY title` orders by each row's title value
- `ORDER BY 'title'` orders using the constant literal `'title'` for each row – same value for each row means no sorting

Stopping Injection Isn't Enough

- injection defenses stop input from changing the query – they don't check whether a value makes sense
- some values are wrong for the field
  - e.g. a replacement cost of -50.00, an ISBN of "hello", a 10,000 character title
- bad values can cause problems beyond the database too
  - e.g. a photo URL starting with `javascript:`, a book title containing `<script>`
- database constraints can catch these if they are declared, but
  - the client gets a generic error ("invalid value" or 500 Internal Server Error), not which field or why
  - policy rules (e.g. max replacement cost) belong in the service, not the schema
- validation in the service allows for specific error messages, before any work is done
  - database constraints remain the backstop

Allow-Lists and Block-Lists

- an *allow-list* defines what is acceptable, everything else is rejected
  - e.g. `sort` is `title`, `author`, or `published`; an ISBN is 10 or 13 digits; a URL starts with `http://` or `https://`
- a *block-list* defines what is forbidden, everything else is accepted
  - e.g. reject input containing `'`, `--`, or `DROP`
- prefer allow-lists
  - what is valid can be listed, every possible attack can't
  - allow-lists describe the field (e.g. what a valid ISBN looks like), not the database
- block-lists can also reject legitimate data
  - e.g. the author O'Brien, a book titled "The Drop"
  - don't block characters that are special in SQL syntax – placeholders already make them harmless

Input Validation

- best practices
  - check type, format, length, range, and required fields as allow lists
  - do the check on the server (in the service layer) so every route gets the same rules
  - invalid input gets a 400 Bad Request that names the field and the problem
- examples
  - an ISBN is 10 or 13 digits – structural, so backed up by CHECK in the DB
  - a title is at most 255 characters – matches the VARCHAR(255) column
  - a replacement cost is between 0 and 500.00 – upper limit is policy, so only check by the application
  - `role` is either `member` or `librarian` –

Handling Input

- always use both placeholders and validation
  - each covers what the other can't

Auditing

- book 42's replacement cost changed from 20.00 to 200.00 and a member was billed 200.00 for losing it
  - who changed it?
  - when?
  - from where?
  - what was it before?
- *auditing* records who did what so you can answer those questions later
  - accountability – people know their changes are recorded
  - detection – unusual activity shows up
  - investigation / forensics – reconstructing what happened after an incident

Audit Trails

```
CREATE TABLE BOOK_AUDIT (
  audit_id    INT          NOT NULL AUTO_INCREMENT,
  book_id     INT          NOT NULL,
  changed_by  INT          NOT NULL,  -- librarian's member_id, from the session
  changed_at  DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP,
  client_ip   VARCHAR(45)  NULL,
  old_cost    DECIMAL(6,2) NULL,
  new_cost    DECIMAL(6,2) NULL,
  PRIMARY KEY (audit_id)
);
```

- an *audit trail* captures who, what, when, from where, and before and after
- write the audit row in the same transaction as the change so both change and audit info commit or neither does

Generating Audit Records

- application writes audit rows or log entries
  - application has to record
  - application knows the app user
- trigger on the table
  - database records automatically
  - database doesn't know the app user itself – must be passed from the application
- DBMS audit features
  - records statements per database account
  - doesn't know the app user

Protecting the Audit Trail

```
GRANT INSERT ON demo_library.BOOK_AUDIT TO 'library_app'@'localhost';
GRANT SELECT ON demo_library.BOOK_AUDIT TO 'auditor'@'localhost';
```

- audit trail is only useful if it can't be quietly edited
  - application can add rows but not change or delete
- audit data is sensitive itself
  - records who did what, and often personal data
- policy: decide how long to keep the data and who may read it