Part 2 Architectural Task – System Design

1 Design an End-to-End Migration Strategy

Pre-Migration

Before starting it would be good to run scripts to find the broken data. E.g., users in the user table that don't have matching joins in the other services or vice versa. If possible, fix and clean data.

Make a list of all services that will be affected – internal, external, webhooks etc

Take database backups if not automatically setup.

Consider temporarily disabling the ability to change username during this whole process.

Make sure there are e2e/integration tests that can be run at different stages to confirm that everything is still working.

Get baseline performance metrics.

Updates to User and User Roles tables

Create the new user_id field on the user table, it should be unique. Then generate uuids for each row. This gives us the mapping from username -> uuid

Then create the new user_roles table.

Create compatibility views

This should be non-breaking as just an addition of a column – dual key setup in transition period

Update dependent tables/services

Add new field user_id to each of the dependent tables. Use the username to backfill the uuid from the user table.

Create compatibility views.

Update dependent service logic to use the new user_id field.

Add foreign key constraints on user_id.

Create new indexes for these tables using the user_id.

Deploy one updated service at a time using feature flags if needed.

External API

Version the api so that the current version (v1) continues to work as is.

Create a new version (v2) that uses the new user_id (breaking changes).

If needed, update v1 (v1.1) to return both username and user_id in case external services need the user_id before fully migrating to the new api.

Post migration

Run tests (see testing section) to confirm everything is still working.

Remove username from dependent service tables.

Check database performance compared to before.

Remove compatibility views.

Update logic to now allow users to access multiple accounts (using the new user_roles table)

Re-enable the ability for users to update their usernames

2. Identify Database Migration Steps

- Update user table with new user_id field, both user_id and username need to be unique.
- Generate uuid for existing users
- Create user_roles table
- Add new user_id field to dependent tables
- Create compatibility views if needed

- Backfill dependent tables with user_id mapping from current username
- Update primary key / foreign key constraints to use new user_id
- Create new indexes for tables using new user_id

3. Service Adaptation Plan

Internal Services

Can gradually swap internal services over to use new user_id, making all changes in development environment first.

Will need to continue to write to both uuid & username.

Assuming username is used when logging in and won't be removed.

Compatibility layer can be used to handle both username and user_id queries.

External Services

Release new version of api v2, whilst continuing to maintain v1 for legacy support. Give clear date for when v1 will be deprecated.

Needs to be clear communication about changes being made and updates to documentation/swagger, release notes, breaking changes etc

4. Risk Mitigation

All changes should be made on a dev environment and tested before moving to production.

Migrations of services should be done outside regular work hours if possible.

Should be ready to revert any changes. Should have detailed monitoring setup to check error rates before/after changes.

Can update service by service one at a time to isolate changes making testing each stage easier.

Canary deployments where we can let a small percentage of traffic use the updated system, slowly increasing whilst monitoring for errors.

Monitoring on performance before/after. Creating compatibility layers may cause potential slow down.

Should have data validation tests where we can compare the results of searching for user via username and user_id.

Use transaction where possible while making database updates/backfilling to stop partial updates.

Rollback options

We should be able to easily rollback in case of any problems.

Data – should be able to restore to a backup of the database
Schema – should be able to drop the new fields and restore the previous constraints
Dependent services – for each dependent service, we should be able to roll back to a previous deployment

We should be able to make the changes in a dev environment, then test our rollback strategy to make sure, everything goes back to a working state.

Can trigger the rollback manually or automatically if there is an increase in errors on the canary deployment.

5. Testing & Verification

Initial checks to find broken data, missing joins.

Shadow reads fetching by username and user_id in parallel, comparing the results

Performance/load testing

Post checks to find any broken data, missing joins