Tennis Tournament Management System – Assignment 2

Crihălmeanu Tudor – 30434

Scope of the Second Iteration

Assignment 2 extends the first release in four directions. First, the player registration flow now involves an explicit approval or denial by an administrator. Second, an e-mail notification is sent to the player once the decision has been taken. Third, the simple player search evolved into a filtering endpoint that understands a keyword, an optional tournament id and a *has-unfinished-match* flag. Fourth, Spring Security now guards every REST endpoint so that only users with the correct role can reach it. New JUnit/Mockito suites accompany every behaviour change.

Layered Architecture

The project keeps the classical three–layer Spring pattern. Controllers stay lightweight REST adapters that only validate input and delegate to services. Services encapsulate all business logic, while repositories extend `JpaRepository` and hide persistence details. A dedicated `SecurityConfig` class adds the Spring Security filter chain in front of the controllers; BCrypt remains the password encoder. `EmailService` lives in the service layer and relies on a Spring-managed `JavaMailSender`. Filtering logic is implemented in `UserService.filterPlayers`; it first calls a JPQL query for coarse selection and then refines the result in memory with the help of `MatchRepository`.

Mailing Workflow

When an administrator approves or denies a request, `RegistrationRequestService` calls `EmailService.sendRegistrationDecision`. That method builds the subject and body, wraps them into a `SimpleMailMessage`, and delegates to the injected `JavaMailSender`. During unit tests the sender is replaced by a Mockito mock; the test asserts that exactly one message, with the expected subject and body, is dispatched.

Advanced Player Filtering

The endpoint `/users/search` receives a `PlayerFilterDTO`. A single JPQL query applies the keyword and the tournament filter while guaranteeing that only players are returned. If the DTO specifies the *has-unfinished-match* predicate, `UserService` looks up player ids produced by the native query `MatchRepository.findPlayerIdsWithIncompleteMatches` and keeps or removes users accordingly. The controller finally returns one JSON array with the matching players.

HTTP Security Rules

`SecurityConfig` disables CSRF, whitelists public paths (`/auth/*`, static front-end files) and protects every remaining URI with `hasRole(...)` expressions that follow the PDF specification delivered with the assignment. Because the front-end stores the user id and role in `localStorage`, stateful sessions are not required. Security unit tests rely on `@WithMockUser` to prove that a referee cannot access an admin-only endpoint and that the expected status codes are returned.

Unit testing (service layer)

Only the service layer is covered by automated unit tests for this iteration, because it concentrates all business rules that must behave deterministically regardless of environment. Every test is written with JUnit 5, Mockito and AssertJ; data-access collaborators are mocked so that no database nor SMTP server is required.

MatchServiceTest verifies that a referee cannot update a match that is assigned to someone else, guaranteeing the integrity of the score-editing rule.

RegistrationRequestServiceTest covers both the happy-path of approving a request—status update, tournament membership side-effect, e-mail dispatch—and the guard that prevents duplicate requests.

TournamentServiceTest demonstrates that only players (not referees or admins) may self-register for tournaments and that a successful registration persists the new association.

UserServiceTest focuses on authentication logic: it stubs a stored hash and checks that a valid password is accepted and the correct `Optional<User>` is returned.

EmailServiceTest captures the `SimpleMailMessage` sent through the mock `JavaMailSender` and asserts that the recipient, subject and body contain all dynamic parts in both the approval and denial scenarios.

Together these focused tests guarantee that the core business behaviours introduced or modified in A2 remain stable as the code evolves.

Diagrams

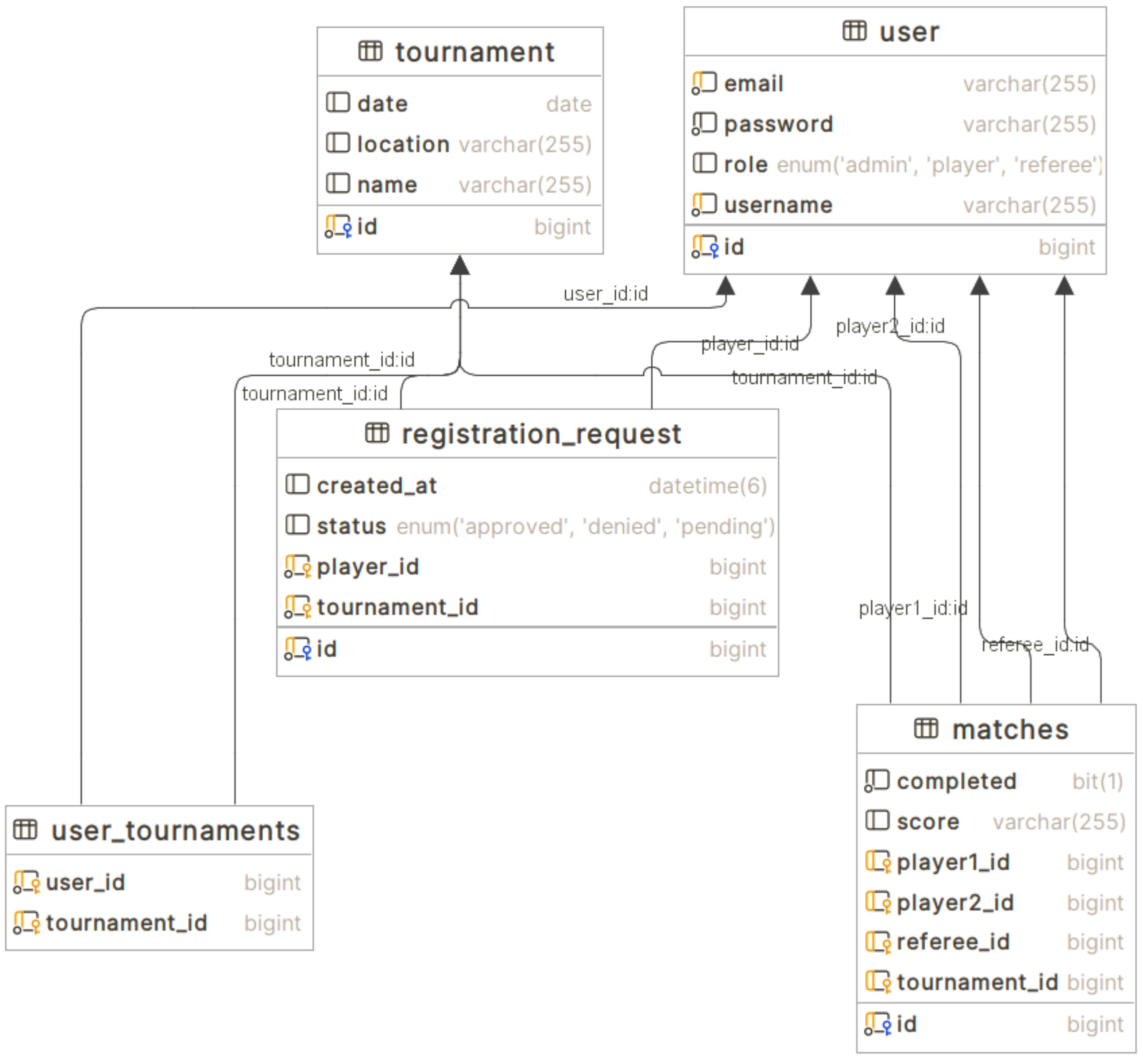

Figure 1: Database Schema

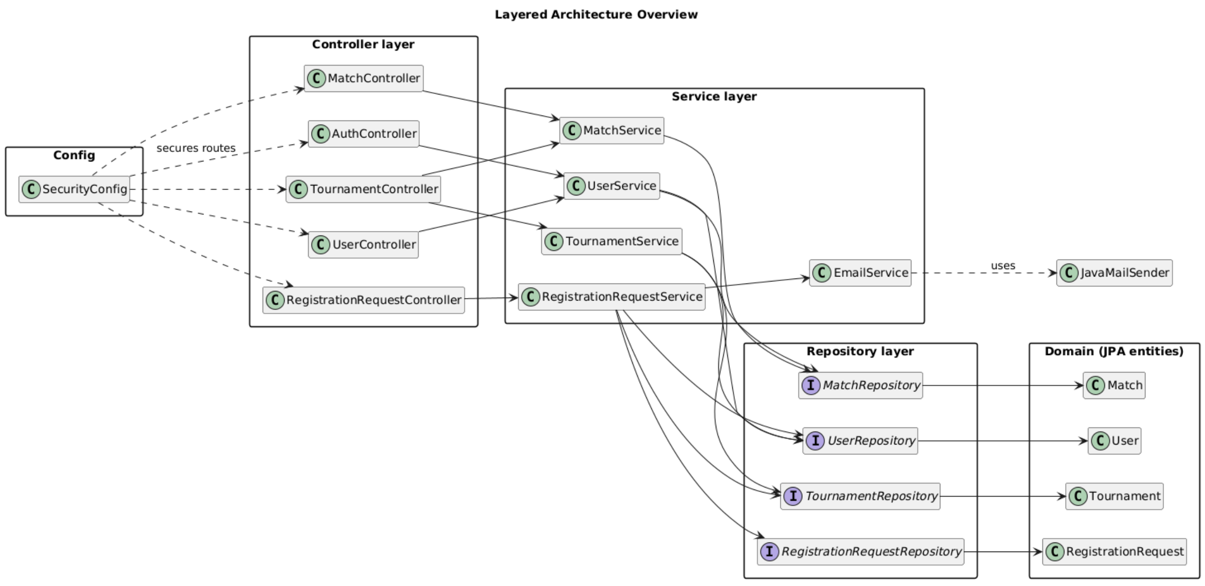

Figure 2: Class Diagram

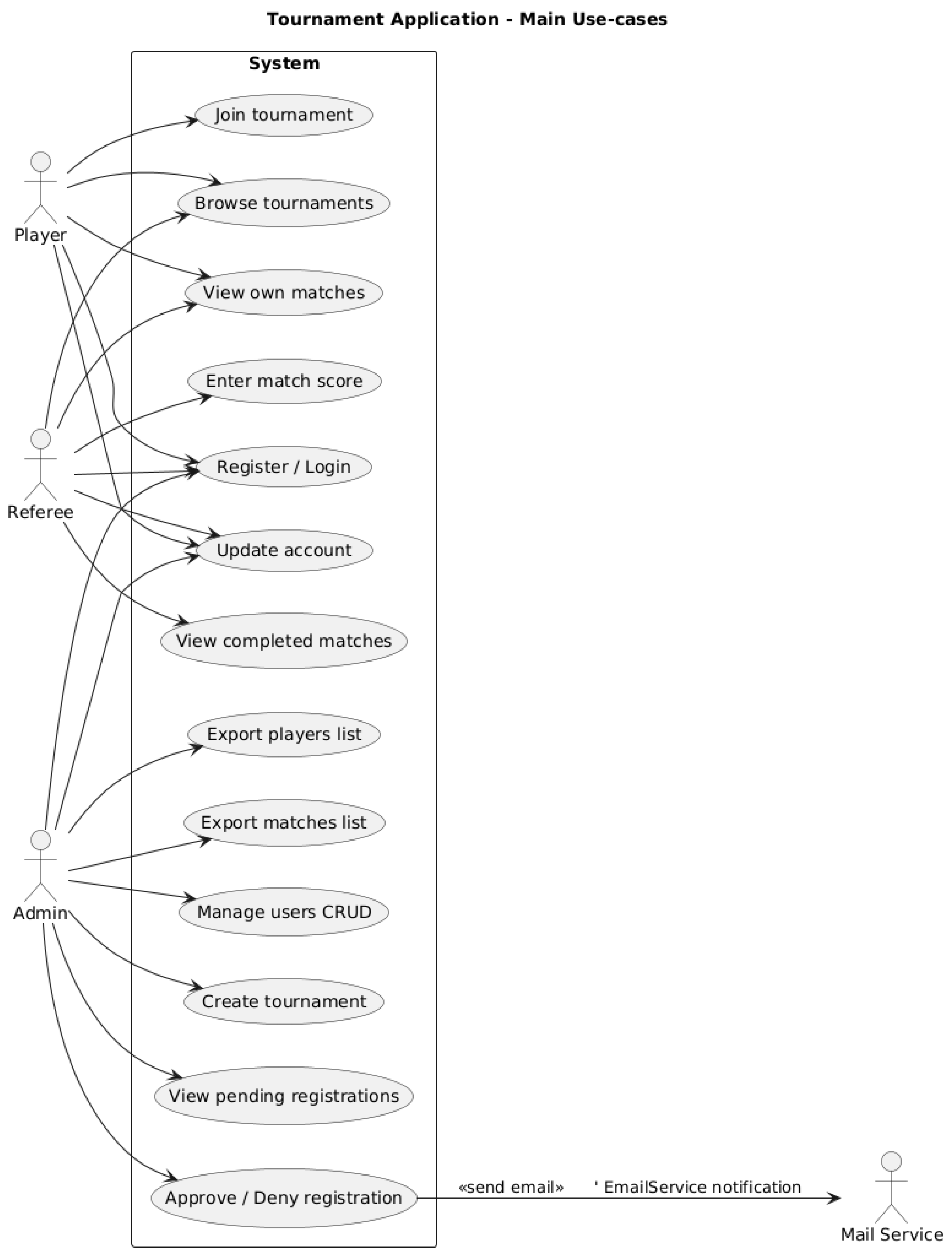

Figure 3: Use Case Diagram