Running Club

Objective:

Develop a **REST API** for a **Running Club** using **Spring Boot**. The system will have two roles: **ADMIN** and **USER**.

Users can register for running events, while admins can create and manage events (though they certainly can register for running events, too).

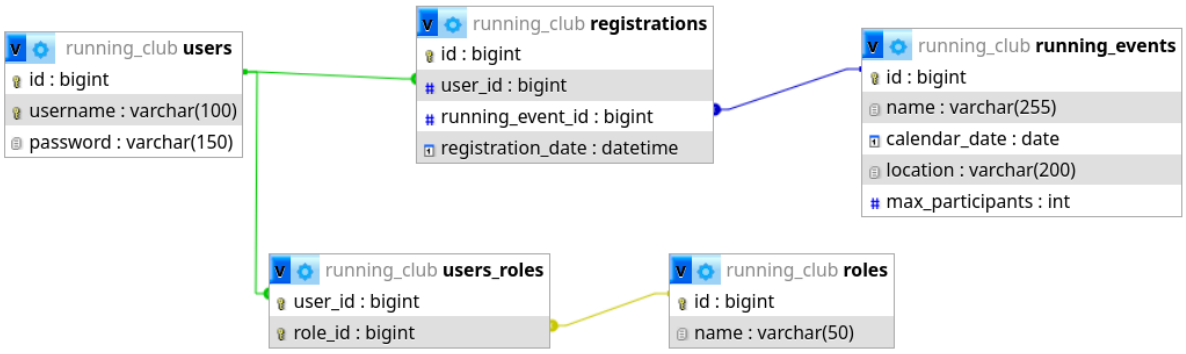
Requirements

Database Tables (3 points)

- In a new database `running_club`, implement `users`, `running_events` and `registrations` tables, with these columns:
- `users`
  - `id`: `BIGINT`, `PRIMARY KEY`, `AUTO_INCREMENT`
  - `username`: `VARCHAR(100)`, `NOT NULL`, `UNIQUE`
  - `password`: `VARCHAR(150)`, `NOT NULL`
- `running_events`
  - `id`: `BIGINT`, `PRIMARY KEY`, `AUTO_INCREMENT`
  - `name`: `VARCHAR(255)`, `NOT NULL`
  - `calendar_date`: `DATE`, `NOT NULL`
  - `location`: `VARCHAR(200)`, `NOT NULL`
  - `max_participants`: `INT`
- `registrations`
  - `id`: `BIGINT`, `PRIMARY KEY`, `AUTO_INCREMENT`
  - `user_id`: `BIGINT`, `FK`
  - `running_event_id`: `BIGINT`, `FK`
  - `registration_date`: `DATETIME`
- You must also add `roles` and `users_roles` tables - you should already know how to do this. This is so a registered person can have more than one role
  - Inside `roles` table, don't forget to manually add two roles: `ROLE_USER` and `ROLE_ADMIN`
- Check the `Designer` in `phpMyAdmin` to see if your database design looks good, has relationships.

- The idea is this: one user can participate in one or many running events. One running event can have one or many participants. The `registrations` table is the junction table for dealing with the many-to-many relationship

How the database design should look:



Entity Models (2 points)

- Create new Spring Boot project, using Spring Initializr. Dependencies:
  - `Spring Web`
  - `Spring Data JPA`
  - `MySQL Driver`
  - `Spring Security`
  - `Validation`
  - `Spring Boot DevTools`
    - Not required; provides hot-reload functionality
- Based on the database and tables you have created just now, create Java `@Entity` classes, in `model` package
  - MySQL to Java Type Mapping
  - You must create `Registration` class too. Why? Because the class has more fields than just the relationship. This is the table for allowing many-to-many relationship
- Entities need to have annotations determining relationship. Classes

`User` and `RunningEvent` must have a `@OneToMany` relationship, for a list of registrations
    - The `Registration` class itself should contain fields which are of type `User` and `RunningEvent` - they need to be `@ManyToOne`
- Create three repositories: `UserRepository`, `RunningEventRepository` and `RegistrationRepository`

DTOs & Validation (4 points)

- Use **DTOs** for transfering payload; use Java `record` classes to create them
- Apply **Bean Validation** annotations for input validation.
    - Do not forget to create an `@ExceptionHandler` annotated method in `GlobalExceptionHandler` class; this way, your `400 Bad Request` responses will contain the correct response message, when DTO validations fail; the code is the same as you used in your previous projects
- Required fields and validations:
    - `UserRequestDTO`:
        * `username`
            · Not null
            · Must be **lowercase letters (numbers allowed), at least 4 characters long**.
        * `password`
            · Not null
            · Must be **at least 6 characters long**.
        * `roles`
            · Not null
    - `UserResponseDTO`:
        * `id`
        * `username`
        * `roles`
    - `RunningEventRequestDTO`:
        * `name`
            · Not null
            · Must be **at least 5 characters long**.
        * `calendarDate`

        - Not null
        - Must be in the **future**.
    * `location`
        - Not null
        - Can only contain letters and numbers.
    * `maxParticipants`
        - Cannot be 0.
- `RunningEventResponseDTO`:
    * `id`
    * `name`
    * `calendarDate`
    * `location`
    * `maxParticipants`
- `RegistrationRequestDTO`:
    * `user`
        - Cannot be null
- `RegistrationResponseDTO`
    * `id`
    * `userId`
        - Notice that `userId` is being returned, not `user`. This means, only the `id` (which is of type `long`) is being returned - not the whole object representing the `user`. This is easier than it sounds.
    * `eventName`
    * `registrationDate`

Security & Authorization (4 points)

- Implement **Basic Authentication** - JWT not required.
- Users should be assigned roles: **USER** or **ADMIN**.
- Only **ADMIN** can create and delete events.
- Regular **USERs** can only register for events.
- Reminder: `Role` implements `GrantedAuthority`. `User` implements `UserDetails`
    - Don't forget to add a `PasswordEncoder` bean. You will also need a separate class that implements `UserDetailsService`
- Implement these `requestMatchers`, in your `SecurityFilterChain`:

  - `POST /api/auth/register` -> allow anyone (unauthenticated)
  - `POST /api/events` -> allow only ADMINs
  - `DELETE /api/events/**` -> allow only ADMINs
    * The `**` means that there can be anything else, in this case, any id
  - `GET /api/events/*/participants` -> allow only ADMINs
    * A single `*` is used because there is another word to the right
  - All the other endpoints -> authenticated

REST API Endpoints (5 points)

- Remember: you will have to use mapper classes with static methods, in order to receive and return data in the form of DTO.
- Test if validation fails with incorrect data, and correct messages are being displayed, for all of the endpoints below
- Reminder: you'll need to create `service` classes, which will communicate with `repository` layer. Controllers use `service` classes

Public Endpoints (non-authenticated)

- `POST /api/auth/register` → Register a new user, `201 CREATED`.
  - `UserRequestDTO` example:

```
{
  "username": "admin",
  "password": "123456",
  "roles": [
    {
      "id": 1
    },
    {
      "id": 2
    }
  ]
}
```

  - `UserResponseDTO` example:

```
{
  "id": 1,
```

```
  "username": "admin",
  "roles": [
    {
      "id": 1,
      "name": null,
      "authority": null
    },
    {
      "id": 2,
      "name": null,
      "authority": null
    }
  ]
}
```

- Implement this in `UserController` class
- Password must be hashed before saving to database
- Validate that the `username` is unique.
  - Derived query needed, inside repository
- Validate that the fields match validation defined in DTO, when error occurs
  - `400 Bad Request` should be returned, with message - this can be implemented by defining a `@RestControllerAdvice`.

ADMIN Role Endpoints

- `POST /api/events` → Create a new event (**Admin only**), `201 CREATED`.
  - `RunningEventRequestDTO` example:

```
{
  "name": "Charity Run",
  "calendarDate": "2025-06-10",
  "location": "Downtown Park",
  "maxParticipants": 150
}
```

  - `RunningEventResponseDTO` example:

```
{
  "id": 1,
```

```
  "name": "Charity Run",
  "calendarDate": "2025-06-10",
  "location": "Downtown Park",
  "maxParticipants": 150
}
```

  - Validate that the fields match validation defined in DTO. If an error happens, should return `400 Bad Request` along with a JSON message.

- `DELETE /api/events/{eventId}` → Delete an event (**Admin only**), `204 NO CONTENT`.
  - Return `404 Not Found` if no event is found by the provided `eventId`.

USER Role Endpoints (authenticated)

- `GET /api/events` → List all upcoming events.
  - `RunningEventResponseDTO` example, `200 OK`:

```
[
  {
    "id": 1,
    "name": "Charity Run",
    "calendarDate": "2025-06-10",
    "location": "Downtown Park",
    "maxParticipants": 150
  },
  {
    "id": 2,
    "name": "Vilnius City Run",
    "calendarDate": "2025-08-15",
    "location": "Centras",
    "maxParticipants": 100
  }
  // ...
]
```

    * Remember: even if the list is empty, it should still return `200 OK` and the empty list.
- `POST /events/{eventId}/register` → Register for an event.

- `RegistrationRequestDTO` example (the event id here was `1`), `201 CREATED`:

```
{
  "user": {
    "id": 7
  }
}
```

- `RegistrationResponseDTO` example:

```
{
  "id": 10,
  "userId": 7,
  "eventName": "Charity Run",
  "registrationDate": "2025-02-13T16:10:31.794678395"
}
```

- A new `Registration` entity is being persisted
- The controller can be defined in the `RunningEventController` class - all controllers of this assignment can be defined in this class, except for `/auth/register`
- If `eventId` cannot be found, don't forget to return `404 Not Found`
- A user cannot register for the same event twice. If this validation fails, return a meaningful JSON message, alongside `400 Bad Request`
  * Remember: a `JSON` message can be easily created by using the `Map` data structure.
- The currently authenticated person can only register for an event himself; he cannot register others.
- Validate that the fields match validation defined in DTO. If an error happens, should return 400 Bad Request along with a JSON message.

---

2 points

- Implement an endpoint `GET /api/events/{eventId}/participants` to list all registered users for an event (**Admin only**).
  - Returns a list of users (I used `eventId` of `1` here):

```
[
```

```
  {
    "id": 7,
    "username": "runner126"
  },
  {
    "id": 19,
    "username": "beastmode5"
  }
  // ...
]
```

- You will need to create separate response DTO to achieve this
- The endpoint can be defined in `RunnningEventController`
- You'll need to add a derived query to get a list of registrations, based on the provided `eventId`, in the `RegistrationRepository`. Then you'll need to map it to a list of users, and return result to the client
- If an event cannot be found by the provided `id`, return `404 Not Found`

Code Quality & Documentation

- Clean, readable code.
- Proper **package structure** (e.g., `controller`, `service`, `repository`, `dto`, `model`).
- You may write inline comments explaining key components.

Submission Guidelines:

- Add your **GitHub repository link**, and do "Turn in"