Ok, analisei a estrutura e o código da pasta main do seu projeto Spring Boot (PumpTrain API). É um projeto bem estruturado, seguindo padrões comuns do Spring com camadas de Controller, Service, Repository, Entidades e DTOs, além de uma configuração de segurança com JWT.

Aqui estão os pontos que considero mais importantes a serem **melhorados ou verificados antes de iniciar o desenvolvimento do Front-end**:

**Pontos Críticos (Bloqueadores Potenciais para o Front-end):**

1. **Configuração de CORS (Cross-Origin Resource Sharing):**
   * **Problema:** Não encontrei nenhuma configuração explícita de CORS no SecurityConfig.java. Se o seu front-end for servido de um domínio ou porta diferente da API (ex: localhost:3000 para front e localhost:8080 para back), o navegador bloqueará as requisições por padrão por segurança.
   * **Ação:** Adicione a configuração de CORS no SecurityConfig. Você pode fazer isso usando http.cors(Customizer.withDefaults()) e definindo um CorsConfigurationSource Bean, especificando quais origens (domínios do front-end), métodos (GET, POST, PUT, DELETE) e cabeçalhos são permitidos. **Isso é essencial para que o front-end possa se comunicar com a API.**
2. **Documentação da API (Swagger/OpenAPI):**
   * **Problema:** Não há indicação de uma ferramenta de documentação automática (como Springdoc OpenAPI).
   * **Ação:** Integre o springdoc-openapi-starter-webmvc-ui. Isso gerará automaticamente uma documentação interativa (Swagger UI) baseada nas suas anotações de controller (@RestController, @GetMapping, DTOs, etc.). **Isso é fundamental para que os desenvolvedores front-end saibam quais endpoints existem, como chamá-los, quais dados enviar e o que esperar como resposta.**

**Pontos Altamente Recomendados (Melhoram a Integração e Robustez):**

1. **Tratamento de Erros Centralizado e Padronizado:**
   * **Problema:** Os controllers (AuthController, WorkoutSessionController, ActivityLogController) capturam exceções e retornam ResponseEntity com status e mensagens. Isso funciona, mas é repetitivo e pode levar a inconsistências.
   * **Ação:** Implemente um GlobalExceptionHandler usando @ControllerAdvice e @ExceptionHandler. Centralize a captura de exceções comuns (EntityNotFoundException, AccessDeniedException, MethodArgumentNotValidException, IllegalArgumentException, etc.) e mapeie-as para respostas HTTP padronizadas (com status code e um corpo JSON consistente, ex: { "timestamp": "...", "status": 404, "error": "Not Found", "message": "Sessão não encontrada...", "path": "/api/workouts/123" }). Isso simplifica muito o tratamento de erros no front-end.
2. **Validação de Entrada (DTOs):**
   * **Problema:** Alguns DTOs (UserRegistrationDto, LoginRequestDto) usam anotações de validação (@NotBlank, @Email, @Size), e os controllers usam @Valid. No entanto, ActivityLogDto, quando usado para criação/atualização (POST /api/workouts/{sessionId}/activities, PUT /api/activities/{activityId}), poderia ter validações mais robustas (ex: exerciseId não pode ser nulo, sets deve ser positivo, talvez alguma validação no formato das strings repetitions e weightKg se possível). WorkoutSessionCreateDto também poderia ter validações se necessário.
   * **Ação:** Revise *todos* os DTOs que são recebidos em @RequestBody e adicione as anotações de validação apropriadas (@NotNull, @Min, @Max, @Pattern, etc.). Certifique-se que @Valid está presente nos parâmetros correspondentes nos controllers. Combine isso com o tratamento de erros centralizado para MethodArgumentNotValidException.
3. **Consistência nos Endpoints de Atividade:**
   * **Problema:** A criação de ActivityLog (POST /api/workouts/{sessionId}/activities) está em WorkoutSessionController, enquanto a atualização e exclusão (PUT /api/activities/{activityId}, DELETE /api/activities/{activityId}) estão em ActivityLogController.
   * **Ação:** Considere unificar a gestão de ActivityLog em um único controller (ActivityLogController). Mesmo o endpoint de criação pode ficar lá, recebendo sessionId como parâmetro ou parte do path, se preferir (ex: POST /api/sessions/{sessionId}/activities ou POST /api/activities com sessionId no corpo - embora o primeiro seja mais RESTful). Isso torna a API mais previsível para o consumidor (front-end).

**Boas Práticas e Melhorias Adicionais:**

1. **Logging:**
   * **Problema:** O uso de Slf4j está comentado nos controllers. Os níveis de log no application.properties estão como DEBUG, o que é bom para desenvolvimento, mas deve ser ajustado para produção.
   * **Ação:** Descomente e utilize um framework de logging (como SLF4j com Logback, que é o padrão do Spring Boot) de forma consistente nos services e controllers para registrar informações importantes, avisos e erros. Isso é vital para depuração e monitoramento.
2. **Testes (Unitários e de Integração):**
   * **Problema:** Não há arquivos de teste no projeto fornecido.
   * **Ação:** Crie testes unitários para os Services (usando Mockito para mockar repositórios) e testes de integração para os Controllers (usando @SpringBootTest e MockMvc ou RestAssured) para garantir que a lógica de negócio e os endpoints funcionam como esperado. **Testes são cruciais para a estabilidade antes de acoplar um front-end.**
3. **Mapeamento Entidade <-> DTO:**
   * **Problema:** O mapeamento é feito manualmente nos services (ex: convertToDto em ExerciseService e WorkoutSessionService). Funciona, mas pode ser verboso e propenso a erros conforme o número de campos aumenta.
   * **Ação:** Considere usar uma biblioteca como MapStruct. Ela gera o código de mapeamento em tempo de compilação com base em interfaces simples, reduzindo boilerplate e aumentando a segurança de tipos. (Como você mesmo comentou no código).
4. **Estrutura de Dados de ActivityLog (repetitions, weightKg):**
   * **Problema:** Armazenar repetitions e weightKg como String (ex: "12,10,8") é flexível, mas dificulta a validação, processamento numérico (ex: calcular volume total) e a consulta no banco de dados.
   * **Ação:** Avalie se essa flexibilidade é realmente necessária. Alternativas:
     + Criar uma entidade SetLog (@OneToMany em ActivityLog) com campos setNumber, repetitions (Integer), weightKg (Double).
     + Usar um tipo JSON no banco (se suportado e desejado) para armazenar uma estrutura mais complexa.
     + Manter como String, mas implementar validação robusta e lógica de parsing/serialização cuidadosa.
5. **Configuração de Produção:**
   * **Problema:** O application.properties usa H2 (banco em memória/arquivo), ddl-auto=update e tem o segredo JWT visível (mesmo que codificado em Base64, não é seguro).
   * **Ação:** Planeje o ambiente de produção:
     + Use um banco de dados persistente (PostgreSQL, MySQL, etc.).
     + Mude ddl-auto para validate ou none e use uma ferramenta de migração (Flyway, Liquibase) para gerenciar o schema do banco.
     + Externalize configurações sensíveis (senha do banco, segredo JWT) usando variáveis de ambiente, Spring Cloud Config, HashiCorp Vault ou similar. **Não comite segredos no código.**

Recomendo focar nos pontos **Críticos** (1, 2) e **Altamente Recomendados** (3, 4, 5) antes de investir pesado no desenvolvimento do front-end, pois eles têm o maior impacto na integração e usabilidade da API. Os demais pontos contribuem para a qualidade e manutenibilidade geral do backend.