### **¿Por Qué Separar los Datos de Usuario?**

1. **Seguridad Mejorada:**
   * **Hashing de Contraseñas:** Las bases de datos relacionales están optimizadas para operaciones criptográficas como el hashing seguro de contraseñas (usando algoritmos como BCrypt, SCrypt o Argon2). Almacenar solo el hash y no la contraseña en texto plano es fundamental. Fuseki/RDF no está diseñado para esto.
   * **Control de Acceso Granular:** Las bases de datos relacionales ofrecen mecanismos robustos para definir permisos específicos sobre tablas y columnas, limitando el acceso a datos sensibles.
   * **Prevención de Ataques Comunes:** Separar las bases de datos reduce el "radio de explosión". Si tu base de datos semántica (Fuseki) se viera comprometida, los datos de autenticación de los usuarios estarían en un sistema diferente y potencialmente más seguro.
   * **Cumplimiento Normativo:** Dependiendo de las leyes de protección de datos (como GDPR), puede ser un requisito o una buena práctica aislar y proteger especialmente los datos personales identificables (PII).
2. **Rendimiento para Autenticación:**
   * Las bases de datos relacionales están altamente optimizadas para búsquedas rápidas de usuarios por nombre de usuario o correo y para la verificación de contraseñas, operaciones comunes durante el login. SPARQL, aunque potente, puede no ser tan eficiente para estas tareas específicas a gran escala.
3. **Integración con Frameworks de Seguridad:**
   * Frameworks como **Spring Security** se integran de forma más natural y directa con bases de datos relacionales para la gestión de usuarios, roles y permisos. Aunque es posible integrarlos con almacenes NoSQL o RDF, suele requerir más configuración personalizada.
4. **Escalabilidad de la Autenticación:**
   * A medida que tu plataforma crezca, el sistema de autenticación basado en una base de datos relacional puede escalar de manera más predecible para un alto volumen de solicitudes de login/registro.

### **¿Qué Implica para tu Proyecto?**

Este cambio es significativo y afectará varias partes de tu aplicación:

* **Modelo de Datos:**
  + Los datos de usuario como nombreUsuario, correo, nombres, apellidos, tieneHashContrasena, fechaCreacion, ultimoAcceso, estaActivo y tieneRol se moverían a la base de datos relacional.
  + La URI del usuario (cerca:usuario\_XXXX) seguiría siendo importante. Podrías almacenarla en la base de datos relacional como un identificador único y también usarla en Fuseki para enlazar con los datos que *sí* permanecen allí (como los anuncios que publica, los intereses que expresa, etc.).
* **Backend (Spring Boot):**
  + **Nueva Dependencia:** Necesitarás añadir dependencias para Spring Data JPA (para interactuar con la BD relacional) y el driver JDBC de la base de datos que elijas (ej. postgresql o mysql-connector-java).
  + **Nuevas Entidades JPA:** Crearás clases Entity (@Entity) para representar a tus usuarios en la base de datos relacional (ej. UserEntity).
  + **Nuevos Repositorios JPA:** Crearás interfaces Repository (@Repository) que extiendan JpaRepository para las operaciones CRUD sobre los usuarios.
  + **Modificación del SparqlQueryService y UserController:**
    - Las funciones de registro (registerUser) y login (loginUser) en SparqlQueryService se modificarán drásticamente o se reemplazarán por lógica que interactúe con la nueva base de datos relacional a través de los repositorios JPA.
    - El UserController llamará a estos nuevos métodos de servicio.
    - La obtención del perfil de usuario (findUserProfileByUri) podría necesitar obtener algunos datos de la BD relacional y otros de Fuseki (si mantienes datos como la calificación promedio o biografía en Fuseki, aunque también podrían migrarse).
  + **Spring Security:** Esta es la parte más importante. Deberás configurar Spring Security para:
    - Usar un UserDetailsService personalizado que cargue los datos del usuario desde tu nueva base de datos relacional.
    - Configurar un PasswordEncoder (ej. BCryptPasswordEncoder) para hashear las contraseñas durante el registro y verificarlas durante el login.
    - Manejar la autenticación y autorización (probablemente con tokens JWT o sesiones).
* **Base de Datos Semántica (Fuseki):**
  + Ya no almacenará contraseñas ni, idealmente, correos electrónicos directamente (a menos que se use como un atributo secundario y no para login).
  + Seguirá almacenando la URI del usuario y las relaciones de ese usuario con anuncios, intereses, etc. La URI del usuario será la "llave" para conectar los datos entre las dos bases de datos.
* **Docker Compose:**
  + Necesitarás añadir un nuevo servicio para tu base de datos relacional (ej. PostgreSQL o MySQL) en docker-compose.yml, con su propia imagen, puertos y volumen para persistencia de datos.
  + El servicio del backend deberá configurarse para conectarse a este nuevo servicio de base de datos.

### **Pasos Generales a Seguir:**

1. **Elegir una Base de Datos Relacional:**
   * **PostgreSQL** es una excelente opción, robusta y de código abierto.
   * MySQL es otra alternativa popular.
2. **Actualizar pom.xml:**
   * Añadir spring-boot-starter-data-jpa.
   * Añadir el driver JDBC para la base de datos elegida (ej. org.postgresql:postgresql).
   * Añadir spring-boot-starter-security.
3. **Configurar la Conexión a la Nueva Base de Datos:**
   * En application.properties (o application.yml), añade las propiedades de conexión para la nueva base de datos (URL, usuario, contraseña, driver).
   * Ejemplo para PostgreSQL:  
     spring.datasource.url=jdbc:postgresql://localhost:5432/nombre\_tu\_bd  
     spring.datasource.username=tu\_usuario\_bd  
     spring.datasource.password=tu\_contraseña\_bd  
     spring.datasource.driver-class-name=org.postgresql.Driver  
     spring.jpa.hibernate.ddl-auto=update # o validate, none en producción  
     spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
4. **Crear la Entidad de Usuario JPA:**
   * Crea una clase User o UserEntity anotada con @Entity y @Table.
   * Define los campos: id (autogenerado), userUri (para enlazar con Fuseki, podría ser UNIQUE), username, email (ambos UNIQUE), password (aquí se almacenará el hash), firstName, lastName, createdAt, lastLogin, isActive, roles (podría ser una relación @ManyToMany a una entidad Role).
5. **Crear el Repositorio de Usuario JPA:**
   * Crea una interfaz UserRepository que extienda JpaRepository<UserEntity, Long>.
   * Define métodos para buscar usuarios, por ejemplo: Optional<UserEntity> findByUsername(String username); y Optional<UserEntity> findByEmail(String email);.
6. **Implementar Hashing de Contraseñas:**
   * Configura un PasswordEncoder bean en tu configuración de Spring Security:  
     @Bean  
     public PasswordEncoder passwordEncoder() {  
      return new BCryptPasswordEncoder();  
     }
   * Usa este passwordEncoder en tu servicio de usuario para codificar la contraseña antes de guardarla y para compararla durante el login.
7. **Modificar Servicios de Registro y Login:**
   * **Registro (UserService o similar, ya no en SparqlQueryService para esta parte):**
     + Recibe el UserRegistrationDto.
     + Verifica si el usuario o correo ya existen usando el UserRepository.
     + Hashea la contraseña.
     + Crea una nueva UserEntity y la guarda usando userRepository.save().
     + **Importante:** También deberás crear la entrada correspondiente en Fuseki con la userUri y los roles básicos (ej. cerca:RolBuscador) para mantener la consistencia con el resto de tu aplicación semántica. Esto podría hacerse llamando a un método específico en SparqlQueryService después de crear el usuario en la BD relacional.
   * **Login (UserService o similar):**
     + Usa Spring Security. Implementa UserDetailsService para cargar el usuario por nombre de usuario/correo desde UserRepository. Spring Security se encargará de comparar el hash de la contraseña.
8. **Configurar Spring Security:**
   * Crea una clase de configuración que extienda WebSecurityConfigurerAdapter (en versiones más antiguas de Spring Boot) o define un SecurityFilterChain bean (en versiones más nuevas).
   * Configura el UserDetailsService y PasswordEncoder.
   * Define qué endpoints requieren autenticación y cuáles son públicos (ej. /api/usuarios/login y /api/usuarios para registro deberían ser públicos).
   * Implementa la gestión de sesiones o tokens JWT.
9. **Actualizar UserController:**
   * Inyecta tu nuevo UserService (o el servicio que maneje la lógica de usuarios con la BD relacional).
   * Modifica los endpoints /login y /register para usar los nuevos métodos del servicio.
   * El endpoint /yo (perfil) ahora obtendrá la información del usuario autenticado a través de Spring Security y podría combinarla con datos de Fuseki si es necesario.
10. **Actualizar SparqlQueryService:**
    * Elimina la lógica de almacenamiento/consulta de contraseñas, correos (para login), etc.
    * Asegúrate de que las consultas que relacionan anuncios, intereses, etc., con usuarios sigan funcionando usando la userUri.
    * Podrías necesitar un método para crear la "contraparte" del usuario en Fuseki (URI y roles) después de que se registre en la BD relacional.
11. **Actualizar docker-compose.yml:**
    * Añade un servicio para tu base de datos relacional (ej. postgres):  
      services:  
       # ... tu servicio backend ...  
       backend:  
       # ...  
       environment:  
       # ...  
       - SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/cercademi\_users # db es el nombre del servicio de la BD  
       - SPRING\_DATASOURCE\_USERNAME=tu\_usuario\_bd  
       - SPRING\_DATASOURCE\_PASSWORD=tu\_contraseña\_bd  
       depends\_on:  
       - fuseki  
       - db # Nueva dependencia  
        
       # ... tu servicio fuseki ...  
        
       db: # Nuevo servicio para PostgreSQL  
       image: postgres:15   
       container\_name: cercademi-postgres-db  
       environment:  
       POSTGRES\_USER: tu\_usuario\_bd  
       POSTGRES\_PASSWORD: tu\_contraseña\_bd  
       POSTGRES\_DB: cercademi\_users  
       ports:  
       - "5433:5432" # Expone el puerto 5432 del contenedor al 5433 del host (o el que prefieras)  
       volumes:  
       - postgres\_user\_data:/var/lib/postgresql/data  
       networks:  
       - cercademi-net  
        
      volumes:  
       fuseki\_data:  
       postgres\_user\_data: # Nuevo volumen para los datos de PostgreSQL
    * Asegúrate de que las variables de entorno del backend apunten al nuevo servicio de base de datos dentro de la red Docker.
12. **Migración de Datos (si aplica):**
    * Si ya tienes usuarios en Fuseki, necesitarás un script o proceso para migrar sus datos relevantes (nombres, correos, generar URIs si no son consistentes) a la nueva base de datos relacional. Las contraseñas no se pueden migrar directamente; los usuarios existentes tendrían que pasar por un proceso de "olvidé mi contraseña" para establecer una nueva contraseña hasheada, o se les podría asignar una contraseña temporal.

### **Consideraciones Adicionales:**

* **Transacciones:** Si una operación implica escribir tanto en Fuseki como en la base de datos relacional (ej. registro de usuario), considera cómo manejar la consistencia. Las transacciones distribuidas pueden ser complejas; una alternativa es asegurar que la operación principal (ej. creación en la BD relacional) sea la primera y, si falla, no se proceda con la escritura en Fuseki.
* **Identificador de Usuario:** La userUri que usas en Fuseki es clave. Asegúrate de que se genere consistentemente y se almacene en la tabla de usuarios de la base de datos relacional para poder enlazar los datos.

Este es un cambio estructural importante, pero aumentará significativamente la seguridad y robustez de la gestión de usuarios en tu aplicación. Te recomiendo abordarlo paso a paso y probar exhaustivamente cada componente.

¿Te gustaría que profundice en alguno de estos pasos o que te ayude con la estructura de código para alguna parte específica (por ejemplo, la entidad User JPA o la configuración básica de Spring Security)?