### **Descripción General del Proyecto**

**"CercaDeMi U Rentals"** es una plataforma web diseñada para facilitar el alquiler de viviendas (apartamentos, casas, habitaciones) cerca de puntos de interés, con un enfoque especial en estudiantes universitarios en Florencia, Caquetá. La aplicación permite a los usuarios:

* **Buscar y filtrar** anuncios de viviendas según diversos criterios (precio, tipo, ubicación, comodidades, proximidad a puntos de interés).
* **Ver detalles** de los anuncios, incluyendo información de la vivienda, del proveedor, y su ubicación en un mapa interactivo.
* **Registrarse e iniciar sesión** para acceder a funcionalidades personalizadas.
* **Publicar anuncios** de viviendas (si son proveedores).
* **Gestionar sus propios anuncios** (editar, eliminar, cambiar estado).
* **Marcar anuncios como interesantes** y ver su lista de intereses.
* **Ver y editar su perfil de usuario**.

### **Arquitectura y Funcionamiento**

El proyecto sigue una arquitectura de tres capas principales:

1. **Frontend (Interfaz de Usuario):**
   * Construido con **HTML, CSS y JavaScript (vanilla JS)**.
   * Se encarga de la presentación visual y la interacción con el usuario.
   * Utiliza **Leaflet.js** para la visualización de mapas.
   * Se comunica con el Backend a través de una **API REST**.
   * La interfaz es servida estáticamente por el backend de Spring Boot.
2. **Backend (Lógica de Aplicación):**
   * Desarrollado en **Java** con el framework **Spring Boot**.
   * Expone una **API REST** para que el frontend consuma los datos y realice acciones.
   * Se conecta a una base de datos semántica (Apache Jena Fuseki) para almacenar y recuperar toda la información.
   * Maneja la lógica de negocio, como la creación de consultas SPARQL, el procesamiento de datos de formularios, y la gestión de usuarios (aunque la seguridad avanzada como hashing de contraseñas y JWT está pendiente).
3. **Base de Datos Semántica (Persistencia):**
   * Utiliza **Apache Jena Fuseki** como servidor SPARQL y almacén de datos RDF.
   * Los datos se modelan según una **ontología OWL** (definida en cerca\_individuos\_v3.ttl y qwe.txt) que describe las entidades (Usuarios, Viviendas, Anuncios, Puntos de Interés, Comodidades, etc.) y sus relaciones.
   * Toda la información de la aplicación (usuarios, anuncios, viviendas, etc.) se almacena como tripletas RDF y se consulta mediante **SPARQL**.

**Flujo General de Datos:**

1. El usuario interactúa con el **Frontend** (ej. realiza una búsqueda).
2. El **JavaScript** del frontend realiza una petición HTTP (GET, POST, PUT, DELETE) a un endpoint específico de la **API REST del Backend**.
3. El **Controlador Spring Boot** correspondiente en el backend recibe la petición.
4. El Controlador llama al SparqlQueryService.
5. El SparqlQueryService construye y ejecuta una **consulta SPARQL** (SELECT para leer, INSERT/DELETE/UPDATE para modificar) contra el servidor **Apache Jena Fuseki**.
6. **Fuseki** procesa la consulta sobre el dataset RDF y devuelve los resultados (si es una consulta SELECT).
7. El SparqlQueryService procesa los resultados de Fuseki y los transforma en **DTOs (Data Transfer Objects)**.
8. El **Controlador** devuelve los DTOs como una respuesta JSON al **Frontend**.
9. El **JavaScript** del frontend recibe la respuesta JSON y actualiza dinámicamente el **HTML** para mostrar la información al usuario o confirmar una acción.

**Entorno de Desarrollo:**

* Se utiliza **Docker y Docker Compose** para orquestar los servicios del backend y Fuseki, asegurando un entorno de desarrollo consistente y aislado.
* El backend corre en el puerto 8082 y Fuseki en el puerto 3333 (interfaz web) y 3030 (endpoint SPARQL interno a Docker).
* Se usa un volumen nombrado (fuseki\_data) para la persistencia de los datos de Fuseki.

### **Análisis Detallado del Código**

#### **1. Ontología y Datos RDF (cerca\_individuos\_v3.ttl, qwe.txt)**

* **qwe.txt (Contiene la definición de la Ontología en formato RDF/XML):**
  + Define las **clases** principales del dominio: Anuncio, Usuario, Vivienda, PuntoDeInteres (con subclases como Universidad, LugarReferencia), Comodidad, Calificacion, Mensaje, Rol, etc.
  + Define las **propiedades de objeto** (relaciones entre clases): esPublicadoPor, describeVivienda, incluyeComodidad, tieneRol, expresaInteres, etc. Se definen también sus inversas.
  + Define las **propiedades de datos** (atributos de las clases): tieneTitulo, tienePrecioMonto, tieneLatitud, tieneNombreUsuario, permiteMascotas, fechaCreacionAnuncio, etc., especificando sus dominios (la clase a la que pertenecen) y rangos (el tipo de dato XSD, como xsd:string, xsd:float, xsd:boolean, xsd:dateTime).
  + La versión 3 de la ontología simplifica las reglas, poniéndolas como propiedades directas del Anuncio.
* **cerca\_individuos\_v3.ttl (Instancias/Individuos en formato Turtle):**
  + Contiene los **datos de ejemplo** que pueblan la ontología.
  + Define individuos para PuntoDeInteres (Universidad de la Amazonia, C.C. Gran Plaza, etc.) con sus coordenadas.
  + Define individuos para Comodidad (WiFi, Parqueadero, Amoblado, etc.).
  + Define una gran cantidad de Usuarios de ejemplo con nombres, correos, roles (Buscador, Proveedor), fechas de creación/acceso y, en algunos casos, calificaciones.
  + Define numerosas Viviendas con direcciones, barrios, coordenadas, tipo de vivienda, y número total de habitaciones/baños.
  + Define Anuncios, vinculándolos a un Usuario (proveedor) y una Vivienda. Cada anuncio tiene título, precio, término de contrato, si es compartido, habitaciones disponibles, fecha de disponibilidad, estado, reglas (mascotas, fumar, etc.) y las comodidades que incluye.

#### **2. Configuración del Entorno (docker-compose.yml, application.properties, pom.xml)**

* **docker-compose.yml:**
  + Orquesta dos servicios: backend (la aplicación Spring Boot) y fuseki (el servidor de base de datos semántica).
  + **Servicio backend:**
    - Se construye a partir de un Dockerfile (no provisto, pero referenciado en el informe de estado).
    - Expone el puerto 8082 del contenedor al 8082 del host.
    - Define variables de entorno para la conexión a Fuseki: FUSEKI\_QUERY\_ENDPOINT, FUSEKI\_UPDATE\_ENDPOINT, FUSEKI\_USERNAME, FUSEKI\_PASSWORD.
    - Depende del servicio fuseki (se iniciará después de Fuseki).
    - Se conecta a la red cercademi-net.
  + **Servicio fuseki:**
    - Utiliza la imagen semapps/jena-fuseki-webacl:latest.
    - Se ejecuta como usuario root (mencionado en el informe para evitar problemas de permisos).
    - Expone el puerto 3030 del contenedor (endpoint SPARQL) al 3333 del host.
    - Utiliza un volumen nombrado fuseki\_data para persistir los datos de la base de datos RDF.
    - Define la variable de entorno ADMIN\_PASSWORD para configurar la contraseña de administrador de Fuseki.
    - Se conecta a la red cercademi-net.
  + Define el volumen fuseki\_data y la red cercademi-net.
* **application.properties:**
  + Configura el puerto del servidor Spring Boot (server.port=8082).
  + Define las propiedades para la conexión a Fuseki cuando se ejecuta localmente (fuera de Docker, o si el backend se conectara al Fuseki expuesto en el host): FUSEKI\_QUERY\_ENDPOINT, FUSEKI\_UPDATE\_ENDPOINT, FUSEKI\_USERNAME, FUSEKI\_PASSWORD. Estas deben coincidir con las variables de entorno usadas en docker-compose.yml cuando se ejecuta en Docker.
* **pom.xml:**
  + Archivo de configuración de Maven para el proyecto backend.
  + Define el spring-boot-starter-parent (versión 3.4.5).
  + Especifica la versión de Java (17).
  + Incluye dependencias clave:
    - spring-boot-starter-web: Para construir aplicaciones web y API REST.
    - lombok: Para reducir código boilerplate (getters, setters, etc.).
    - jena-arq: Para el motor de consultas SPARQL de Apache Jena.
    - jena-rdfconnection: Para conectarse a endpoints SPARQL remotos (como Fuseki).
    - httpclient (Apache HttpComponents): Usado por jena-rdfconnection (aunque Jena también puede usar el HttpClient de Java SE). El informe menciona que se añadió explícitamente para resolver problemas de compatibilidad.
    - jakarta.annotation-api: Para anotaciones estándar de Java EE/Jakarta EE.
  + Configura el plugin de compilación de Maven y el plugin de Spring Boot para empaquetar la aplicación.

#### **3. Backend - Lógica Principal (Java/Spring Boot)**

* **CercaDeMiURentalsApiApplication.java:**
  + Clase principal que arranca la aplicación Spring Boot (@SpringBootApplication).
  + Contiene la configuración **CORS (WebMvcConfigurer)** para permitir peticiones desde cualquier origen (allowedOrigins("\*")) a los endpoints bajo /api/\*\*, utilizando los métodos HTTP comunes. Esto es crucial para que el frontend (servido, por ejemplo, desde file:// o un servidor de desarrollo diferente) pueda comunicarse con el backend.
* **SparqlQueryService.java:**
  + Es el **corazón de la lógica de datos**.
  + Utiliza @Value para inyectar las URLs de los endpoints de Fuseki y las credenciales desde application.properties (o variables de entorno en Docker).
  + **Inicializa un java.net.http.HttpClient con autenticación básica** (@PostConstruct) para todas las comunicaciones con Fuseki. Esto fue una corrección clave mencionada en el informe de estado para resolver errores 401.
  + **executeSelectQuery(String queryString, String methodName):** Método genérico para ejecutar consultas SPARQL SELECT. Construye la consulta, la ejecuta contra sparqlQueryEndpointUrl usando el HttpClient autenticado, y procesa el ResultSet para devolver una lista de mapas (donde cada mapa representa una fila de resultados). Incluye logging detallado y manejo de errores.
  + **executeUpdateQuery(String updateString, String methodName):** Método genérico para ejecutar operaciones SPARQL UPDATE (INSERT, DELETE). Construye la UpdateRequest, la ejecuta contra sparqlUpdateEndpointUrl usando el HttpClient autenticado. Incluye logging y manejo de errores.
  + **Métodos de formato (formatXsdString, formatXsdDecimal, etc.):** Utilidades para formatear correctamente los valores Java como literales RDF tipados para incluirlos en las consultas SPARQL (ej., "texto"^^xsd:string, "123.45"^^xsd:decimal).
  + **createAnuncio(AnuncioCreateDto anuncioData):**
    - Genera URIs únicas para el nuevo anuncio y su vivienda asociada.
    - Construye una consulta SPARQL INSERT DATA compleja para añadir todas las tripletas RDF que describen el nuevo anuncio y la vivienda, incluyendo sus propiedades, relaciones con el proveedor, y comodidades.
    - Utiliza los métodos de formato para asegurar que los datos se inserten correctamente.
  + **updateAnuncio(String anuncioUri, AnuncioUpdateDto anuncioData):**
    - Construye una consulta SPARQL DELETE { ... } INSERT { ... } WHERE { ... }.
    - Primero, busca la URI de la vivienda asociada al anuncio.
    - Luego, para cada propiedad del anuncio y de la vivienda que se puede actualizar:
      * Añade una cláusula DELETE para eliminar el valor antiguo (usando una variable como ?oldTitulo).
      * Añade una cláusula INSERT para añadir el nuevo valor (si se proporciona).
      * Añade una cláusula OPTIONAL en el WHERE para obtener el valor antiguo.
    - Maneja la actualización de las comodidades eliminando todas las antiguas y añadiendo las nuevas.
    - Incluye la actualización de fechaUltimaActualizacion.
  + **registerUser(UserRegistrationDto userData):**
    - Genera una URI única para el nuevo usuario.
    - Construye una consulta SPARQL INSERT DATA para crear el nuevo usuario con sus propiedades (nombre, correo, contraseña - **actualmente en texto plano**, rol por defecto Buscador, etc.).
  + **loginUser(LoginRequestDto loginData):**
    - Construye una consulta SPARQL SELECT para buscar un usuario activo por correo o nombre de usuario.
    - Recupera la URI del usuario y su contraseña almacenada.
    - **Realiza una comparación de contraseñas en texto plano (marcado como inseguro en el informe y necesita hashing).**
    - Si el login es exitoso, actualiza ultimoAcceso del usuario.
  + **findAvailableAnuncios(...):**
    - Construye una consulta SPARQL SELECT DISTINCT para buscar anuncios.
    - Añade patrones WHERE y cláusulas FILTER dinámicamente según los parámetros de búsqueda proporcionados (texto, tipo de vivienda, precio, habitaciones, comodidades).
    - Si se proporcionan pdiUri y distanciaMaxKmFilter, primero obtiene todos los anuncios que cumplen los otros criterios, y luego filtra en Java calculando la distancia Haversine entre cada anuncio y el PDI seleccionado.
  + **findPuntosDeInteres():** Obtiene todos los PDI (Universidades, Lugares de Referencia, Empresas) con sus nombres y coordenadas.
  + **findAnuncioDetailsByUri(String anuncioUri):** Obtiene todos los detalles de un anuncio específico, incluyendo información de la vivienda, proveedor y una lista concatenada de comodidades (que luego se parsea).
  + **findAnunciosByProviderUri(String providerUri):** Obtiene un resumen de los anuncios publicados por un usuario específico.
  + **findUserProfileByUri(String userUri):** Obtiene los datos del perfil de un usuario.
  + **deleteAnuncioByUri(String anuncioUri):** Elimina todas las tripletas donde el anuncio es sujeto u objeto.
  + **updateAnuncioState(String anuncioUri, String nuevoEstado):** Actualiza la propiedad cerca:tieneEstadoAnuncio de un anuncio.
  + **markAnuncioAsInterested(String userUri, String anuncioUri):** Crea una tripleta userUri cerca:expresaInteres anuncioUri.
  + **findInterestedAnunciosByUri(String userUri):** Obtiene los anuncios que un usuario ha marcado como interesantes.
  + **Métodos de extracción (extractStringValue, extractIntegerValue, etc.):** Utilidades para convertir los literales RDF (que vienen como strings desde Jena, ej. "texto"^^xsd:string) a los tipos de datos Java correspondientes.
  + **escapeSparqlLiteral(String literal):** Escapa caracteres especiales en los literales para evitar errores SPARQL.
* **Controladores (AnuncioController.java, UserController.java, TestController.java):**
  + Usan anotaciones de Spring MVC (@RestController, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PathVariable, @RequestParam, @RequestBody) para definir los endpoints de la API REST.
  + Inyectan (@Autowired) el SparqlQueryService para acceder a la lógica de negocio.
  + **AnuncioController.java:**
    - /api/viviendas (GET): Llama a sparqlService.findAvailableAnuncios con los parámetros de filtro.
    - /api/puntos-de-interes (GET): Llama a sparqlService.findPuntosDeInteres.
    - /api/viviendas/{idAnuncio} (GET): Llama a sparqlService.findAnuncioDetailsByUri.
    - /api/anuncios (POST): Llama a sparqlService.createAnuncio.
    - /api/anuncios/{idAnuncio} (PUT): Llama a sparqlService.updateAnuncio.
    - /api/mis-anuncios (GET): Llama a sparqlService.findAnunciosByProviderUri.
    - /api/anuncios/{idAnuncio} (DELETE): Llama a sparqlService.deleteAnuncioByUri.
    - /api/anuncios/{idAnuncio}/estado (PUT): Llama a sparqlService.updateAnuncioState.
    - /api/anuncios/{idAnuncio}/interes (POST): Llama a sparqlService.markAnuncioAsInterested.
    - /api/mis-intereses (GET): Llama a sparqlService.findInterestedAnunciosByUri.
  + **UserController.java:**
    - /api/usuarios (POST): Llama a sparqlService.registerUser.
    - /api/usuarios/login (POST): Llama a sparqlService.loginUser.
    - /api/usuarios/yo (GET): Llama a sparqlService.findUserProfileByUri (temporalmente usando userUri como parámetro).
  + **TestController.java:**
    - /api/test/count-users (GET): Llama a sparqlService.countUsers para una prueba simple.
  + Manejan las respuestas HTTP usando ResponseEntity para establecer códigos de estado (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error) y cuerpos de respuesta.
  + Incluyen logging básico con SLF4J.
* **DTOs (Data Transfer Objects - paquete com.cercademiurentals.api.dto):**
  + Clases Java simples (POJOs), la mayoría anotadas con @Data de Lombok, que definen la estructura de los datos intercambiados entre el frontend y el backend.
  + AnuncioCreateDto: Para crear anuncios.
  + AnuncioDetalleDto: Para enviar detalles completos de un anuncio. Incluye un DTO anidado ProveedorDto.
  + AnuncioSummaryDto: Para listas resumidas de anuncios (ej. en el mapa o resultados de búsqueda).
  + AnuncioUpdateDto: Para actualizar anuncios.
  + LoginRequestDto: Para las credenciales de inicio de sesión.
  + PuntoDeInteresDto: Para la información de los PDI.
  + UserProfileDto: Para los datos del perfil de usuario.
  + UserRegistrationDto: Para los datos de registro de nuevos usuarios.

#### **4. Frontend (HTML, CSS, JavaScript)**

* **Archivos HTML (index.html, detalle.html, etc.):**
  + Estructura semántica básica de cada página.
  + Enlazan style.css para los estilos y script.js para la funcionalidad.
  + index.html incluye la librería Leaflet para el mapa.
  + Contienen elementos con IDs específicos que script.js utiliza para la manipulación del DOM (ej., anuncios-container, map, formularios, etc.).
  + Los formularios (login-form, register-form, crear-anuncio-form, editar-anuncio-form, filter-form) tienen campos (input, select, textarea, checkbox) con nombres que se usan para recoger los datos en JavaScript.
  + La navegación (site-nav) se actualiza dinámicamente por JavaScript según el estado de login.
* **style.css:**
  + Define una paleta de colores y fuentes usando variables CSS (:root).
  + Incluye un reset básico y estilos globales.
  + Estilos detallados para el encabezado, pie de página, contenedores principales.
  + Estilos para los formularios, botones, tarjetas de anuncios.
  + Estilos específicos para la sección de filtros, la lista de anuncios y el mapa en index.html.
  + Estilos para las páginas de autenticación (login.html, register.html).
  + Estilos para las páginas de perfil (profile.html), mis anuncios (my-anuncios.html) y mis intereses (my-intereses.html).
  + Media queries para hacer la interfaz responsiva en diferentes tamaños de pantalla (tablets, móviles).
  + Pequeñas animaciones (ej. fadeIn para las tarjetas de anuncio).
* **script.js:**
  + **Configuración Global:** backendUrl para la URL del API. Variables para el mapa (map) y el grupo de capas de PDI (pdiLayerGroup).
  + **Manejo de Sesión:**
    - getLoggedInUserUri(), setLoggedInUserUri(uri), logoutUser(): Funciones para gestionar la URI del usuario logueado en localStorage.
    - updateNavigation(): Actualiza dinámicamente los enlaces de navegación (ej. mostrar "Mi Perfil" y "Cerrar Sesión" si está logueado, o "Login" y "Registro" si no). Marca el enlace activo.
  + **Funciones del Mapa y Anuncios (index.html):**
    - initializeMap(): Inicializa el mapa Leaflet, lo centra en Florencia, y añade la capa de OpenStreetMap. Crea pdiLayerGroup.
    - getFilterValues(): Recoge los valores de los campos del formulario de filtros.
    - applyFiltersAndLoadAnuncios():
      * Construye la URL de la API con los parámetros de filtro.
      * Realiza la petición fetch al backend (/api/viviendas).
      * Muestra un indicador de carga.
      * Limpia los marcadores de anuncios anteriores del mapa (excepto los PDI).
      * Renderiza las tarjetas de anuncios en anuncios-container.
      * Añade marcadores al mapa para cada anuncio con su información y enlace a detalle.html.
      * Ajusta los límites del mapa para mostrar todos los marcadores.
    - resetFilters(): Limpia el formulario de filtros y recarga los anuncios.
    - loadAndDisplayPuntosDeInteres(mapInstance): Obtiene los PDI de /api/puntos-de-interes y los añade al pdiLayerGroup en el mapa con un ícono personalizado.
    - populatePdiFilterDropdown(): Carga los PDI en el select del formulario de filtros.
  + **Funciones de Páginas Específicas:**
    - cargarDetalleAnuncio() (detalle.html): Obtiene el ID del anuncio de los parámetros URL, llama a /api/viviendas/{idAnuncio}, y renderiza los detalles completos usando renderAnuncioDetails.
    - renderAnuncioDetails(...): Formatea y muestra toda la información del DTO AnuncioDetalleDto en la página de detalle. Incluye un botón para "Marcar como Interesante".
    - handleCrearAnuncioForm() (crear-anuncio.html): Maneja el envío del formulario de creación. Recoge los datos, los estructura en un AnuncioCreateDto (incluyendo providerUri del usuario logueado), y envía una petición POST a /api/anuncios.
    - loadAnuncioForEdit() (editar-anuncio.html): Obtiene los datos del anuncio a editar y llama a populateEditForm.
    - populateEditForm(...): Rellena el formulario de edición con los datos del anuncio.
    - handleEditarAnuncioForm() (editar-anuncio.html): Maneja el envío del formulario de edición. Recoge los datos, los estructura en un AnuncioUpdateDto, y envía una petición PUT a /api/anuncios/{idAnuncio}.
    - handleLoginForm() (login.html): Maneja el envío del formulario de login. Envía una petición POST a /api/usuarios/login. Si es exitoso, guarda la URI del usuario en localStorage y redirige.
    - handleRegisterForm() (register.html): Maneja el envío del formulario de registro. Envía una petición POST a /api/usuarios.
    - loadUserProfile() (profile.html): Obtiene la URI del usuario logueado, llama a /api/usuarios/yo y muestra la información del perfil. Incluye lógica para mostrar/editar la biografía y previsualizar anuncios/intereses.
    - loadMyAnuncios() (my-anuncios.html): Obtiene los anuncios del usuario logueado (/api/mis-anuncios) y los muestra. Incluye botones para editar, eliminar y cambiar estado.
    - handleDeleteAnuncio(): Envía una petición DELETE a /api/anuncios/{idAnuncio}.
    - handleEditAnuncioState(): Pide un nuevo estado al usuario y envía una petición PUT a /api/anuncios/{idAnuncio}/estado.
    - loadMyInterestedAnuncios() (my-intereses.html): Obtiene los anuncios marcados como interesantes por el usuario (/api/mis-intereses) y los muestra. Incluye botón para quitar interés.
    - handleMarkAsInterested(): Envía una petición POST a /api/anuncios/{idAnuncio}/interes.
    - handleRemoveInterest(): Envía una petición DELETE a /api/anuncios/{idAnuncio}/interes.
  + **Manejo del DOM:** Usa document.getElementById, document.querySelector, createElement, appendChild, innerHTML, etc., para interactuar con la página.
  + **Event Listener Principal (DOMContentLoaded):** Detecta en qué página está el usuario y llama a las funciones de inicialización correspondientes. Siempre actualiza la navegación.
  + **Utilidades:** limpiarTituloAnuncio(), renderBoolean().
* **Imágenes (default-avatar.jpg, default-placeholder.jpg, marker-icon-2x-violet.png, marker-shadow.png):**
  + Usadas como imágenes por defecto en el frontend cuando no hay una imagen específica disponible (para avatares o fotos de anuncios) o para los marcadores del mapa.

#### **5. Documentación y Otros**

* **Informe de Estado\_ Proyecto CercaDeMi U Rentals.docx:**
  + Documento crucial que resume el estado del proyecto en un momento dado (7 de Mayo de 2025).
  + Describe la configuración de Docker, la base de datos Fuseki, el estado del backend y frontend.
  + Detalla el proceso de desarrollo y los problemas superados (muy útil para entender el contexto).
  + Enumera los próximos pasos clave, incluyendo mejoras de seguridad, completar funcionalidades CRUD, refinar búsqueda, mejoras UI/UX, refactorización, pruebas y documentación.
* **HELP.md:**
  + Documentación genérica de Spring Boot y Maven, probablemente generada al crear el proyecto Spring Boot.

### **Funcionalidades Clave Implementadas (según el código y el informe)**

* **Gestión de Usuarios:**
  + Registro de nuevos usuarios (contraseña en texto plano).
  + Inicio de sesión (comparación de texto plano).
  + Visualización de perfil de usuario (conteo de anuncios/intereses, información básica).
  + Edición de biografía (frontend, pendiente integración completa con backend para persistencia).
  + Visualización de preferencias de búsqueda (frontend, pendiente integración completa).
* **Gestión de Anuncios:**
  + Creación de nuevos anuncios (vinculados a un proveedor y una vivienda nueva).
  + Visualización de una lista de anuncios disponibles con filtros básicos (texto, tipo, precio, habitaciones) y avanzados (comodidades, proximidad a PDI).
  + Visualización de detalles completos de un anuncio.
  + Edición de anuncios existentes.
  + Eliminación de anuncios.
  + Cambio de estado de un anuncio.
  + Visualización de "Mis Anuncios" para el usuario proveedor.
* **Interacción con Anuncios:**
  + Marcar un anuncio como "interesante".
  + Ver la lista de "Mis Intereses".
  + Quitar un anuncio de la lista de intereses.
* **Integración de Mapa (Leaflet):**
  + Mostrar anuncios como marcadores en un mapa interactivo.
  + Mostrar Puntos de Interés en el mapa.
  + Filtrar anuncios por proximidad a un PDI seleccionado.
* **Interfaz de Usuario:**
  + Navegación dinámica que cambia según el estado de autenticación.
  + Formularios para las diferentes acciones.
  + Diseño responsivo básico.

### **Puntos Críticos y Próximos Pasos (Reflejados en el Informe y el Código)**

* **Seguridad:** La principal prioridad es implementar el hashing de contraseñas (BCrypt) y un sistema de autenticación/autorización robusto (Spring Security, posiblemente JWT).
* **Persistencia de Cambios en Perfil:** La edición de biografía y preferencias en el perfil necesita ser conectada al backend para guardar los cambios en Fuseki.
* **Completar CRUD:** Asegurar que todas las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para anuncios y otras entidades funcionen de manera robusta y segura.
* **Refinamiento de Consultas SPARQL:** Optimizar las consultas para eficiencia, especialmente con filtros complejos.
* **Mejoras UI/UX:** Pulir el diseño, mejorar la usabilidad y el manejo de errores en el frontend.

En resumen, has construido una base sólida para una aplicación web semántica completa. El backend maneja la lógica de negocio y la comunicación con Fuseki, mientras que el frontend proporciona una interfaz interactiva para los usuarios. La ontología define bien el dominio, y Docker facilita el despliegue del entorno. Los próximos pasos identificados en el informe son lógicos y cruciales para llevar el proyecto a un estado de producción.

Espero que este análisis detallado te sea útil. ¡Estoy aquí si tienes más preguntas o necesitas ayuda con alguna tarea específica!