Universidad La Salle
Facultad de Ingenierías y Arquitectura
Carrera Profesional de Ingeniería de Software

MTD

Tech Boutique

E-commerce con Web Semántica

Curso: Web Semantica

Docente: Marco Camacho Alatrista

Estudiante: Velazco Yana, Andrea Del Rosario

Arequipa, Perú
12 de diciembre de 2025

Índice

1. Objetivo

Desarrollar un sistema de e-commerce inteligente que utilice tecnologías de Web Semántica para proporcionar recomendaciones personalizadas de productos tecnológicos basadas en ontologías OWL y razonamiento semántico, detectando automáticamente compatibilidades y optimizando la experiencia de compra del usuario.

2. Título del Software

SemanticShop - Tech Boutique
E-commerce Inteligente con Web Semántica

3. Tecnologías Utilizadas

El proyecto SemanticShop ha sido desarrollado utilizando las siguientes tecnologías:
Frontend:

- **Framework:** React.js
- **Estilos:** Tailwind CSS
- **HTTP Client:** Axios
- **Gestión de estado:** React Hooks (useState, useEffect)

Backend:

- **Framework:** Spring Boot
- **Lenguaje:** Java
- **Arquitectura:** RESTful API
- **Validación:** Spring Validation

Web Semántica:

- **Ontología:** OWL 2 (Web Ontology Language)
- **Razonador:** HermiT Reasoner
- **Framework:** Apache Jena
- **Consultas:** SPARQL

Base de Datos:

- MySQL / PostgreSQL

4. Requisitos Funcionales

El sistema SemanticShop cumple con los siguientes requisitos funcionales:
RF-01: Registro y autenticación de usuarios con preferencias personalizadas.
RF-02: Catálogo de productos con búsqueda y filtros avanzados (marca, precio, disponibilidad).
RF-03: Sistema de recomendaciones basado en ontologías y razonamiento semántico.
RF-04: Detección automática de compatibilidad/incompatibilidad entre productos.
RF-05: Carrito de compras con validación de stock y gestión de cantidades.
RF-06: Proceso de checkout con información de envío y resumen de compra.
RF-07: Gestión completa de pedidos con estados y opciones de cancelación.
RF-08: Panel de configuración para actualizar preferencias del usuario.

5. Diagramas del Sistema

Los siguientes diagramas representan la arquitectura, funcionamiento y estructura del sistema SemanticShop.

5.1. Diagrama de Componentes

El diagrama de componentes muestra la estructura general del sistema organizada en capas. La arquitectura se divide en cuatro capas principales:

Capa de Presentación: Contiene los componentes React (Dashboard, ProductCatalog, Cart, Checkout, Orders, Recommendations) que interactúan con el usuario.

Capa de Aplicación: Servidor Spring Boot que expone endpoints REST para manejar peticiones HTTP del frontend.

Capa de Lógica de Negocio: Servicios Java que implementan la funcionalidad del negocio (UserService, ProductService, OrderService, RecommendationService, OntologyService).

Capa de Datos: Repositorios JPA para persistencia en base de datos y motor de ontologías para razonamiento semántico.

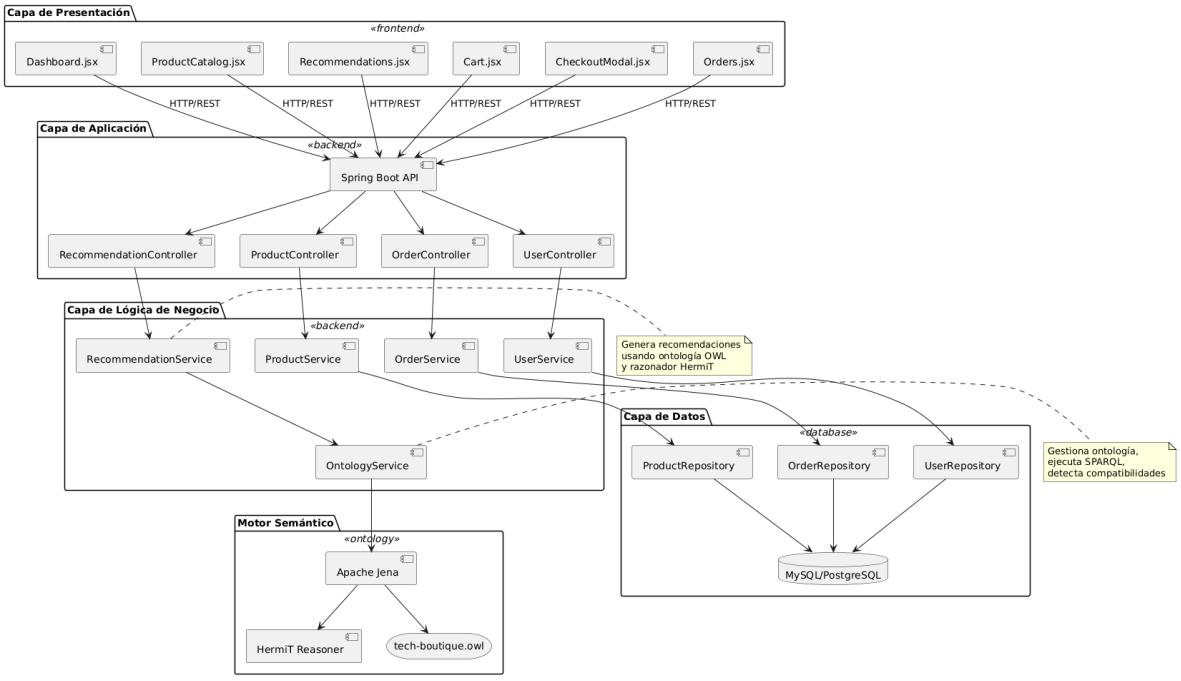

Figura 1: Diagrama de Componentes del Sistema SemanticShop

5.2. Diagrama de Clases

El diagrama de clases presenta la estructura orientada a objetos del sistema, mostrando las entidades principales y sus relaciones.

5.2.1. Módulo de Dominio

Contiene las entidades principales del sistema:

User: Representa un usuario con atributos (id, username, email, password, nombre, telefono, direccion, marcaPreferida, sistemaOperativo, precioMin, precioMax).

Product: Representa un producto con atributos (id, nombre, marca, categoria, precio, stock, descripcion, imagenUrl, tipoConector, sistemaOperativoCompatible).

Order: Representa un pedido con atributos (id, usuario, productos, total, estado, fechaCreacion, direccionEnvio, notas).

OrderItem: Representa un item del pedido (id, producto, cantidad, precioUnitario, subtotal).

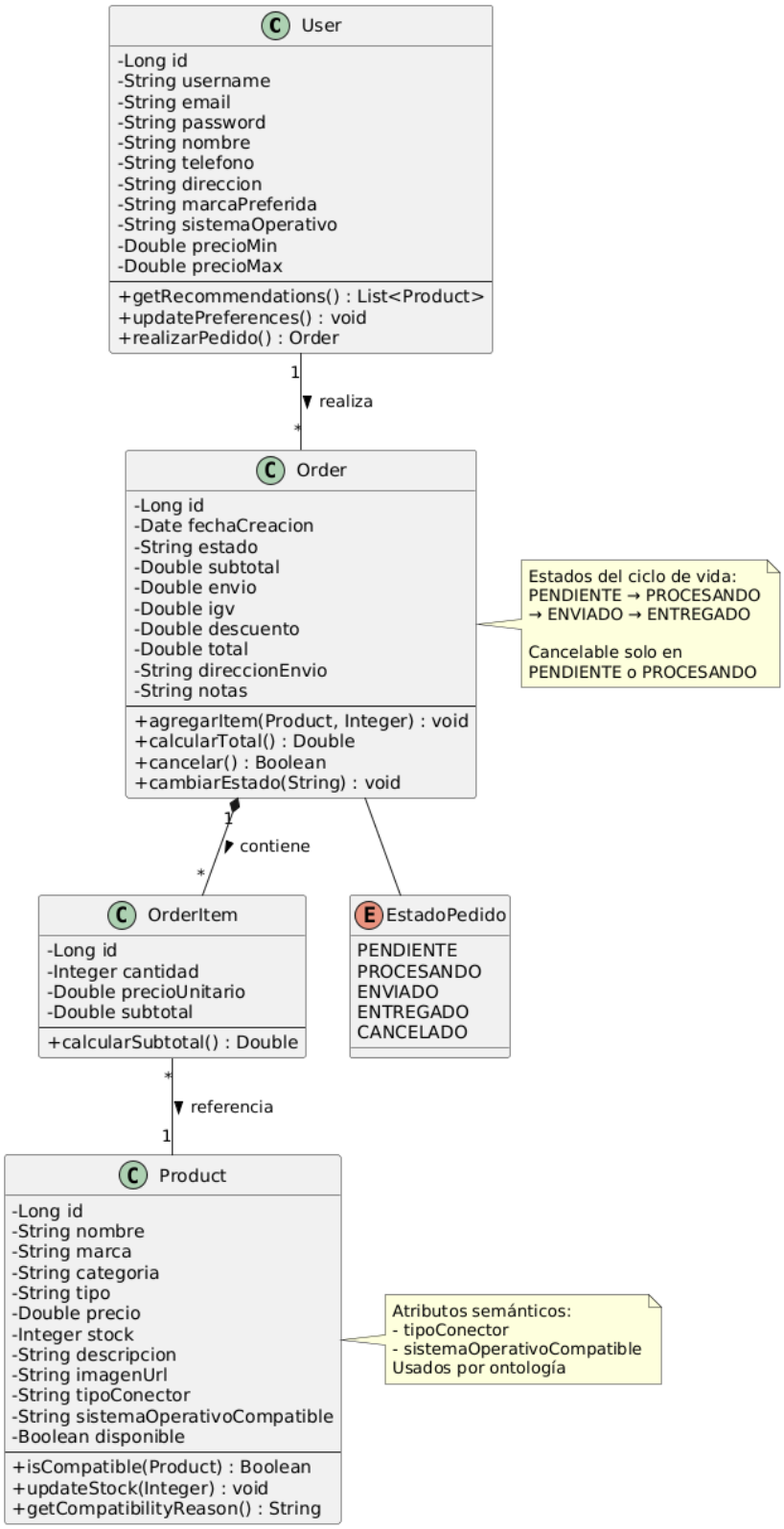

Figura 2: Diagrama de Clases - Entidades de Dominio

5.2.2. Módulo de Servicios

RecommendationService: Genera recomendaciones personalizadas consultando la ontología OWL, ejecutando razonamiento con HermiT y calculando puntuaciones de relevancia.

OntologyService: Gestiona la ontología OWL, ejecuta consultas SPARQL, detecta compatibilidades y mantiene el modelo semántico actualizado.

OrderService: Gestiona el ciclo de vida de los pedidos (crear, actualizar estado, cancelar, listar por usuario).

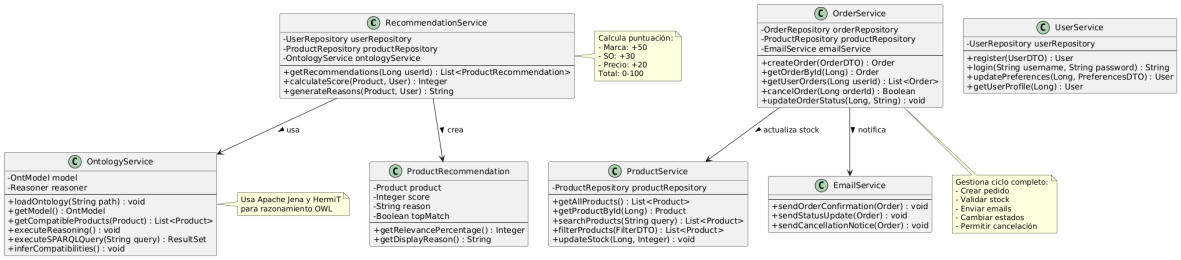

Figura 3: Diagrama de Clases - Capa de Servicios

5.3. Diagramas de Flujo

5.3.1. Flujo: Generación de Recomendaciones

El proceso inicia cuando el usuario accede a la sección de Recomendaciones. El sistema:
Paso 1: Obtiene las preferencias del usuario (marca preferida, sistema operativo, rango de precios).
Paso 2: Consulta la ontología OWL para obtener todos los productos.
Paso 3: Para cada producto, ejecuta el razonador HermiT para inferir compatibilidades.
Paso 4: Calcula puntuación de relevancia basada en:

- Marca coincide: +50 puntos
- SO compatible: +30 puntos
- Dentro de presupuesto: +20 puntos
- Puntuación normalizada: 0-100

Paso 5: Ordena productos por puntuación descendente.
Paso 6: Retorna top 10-20 recomendaciones con razones.

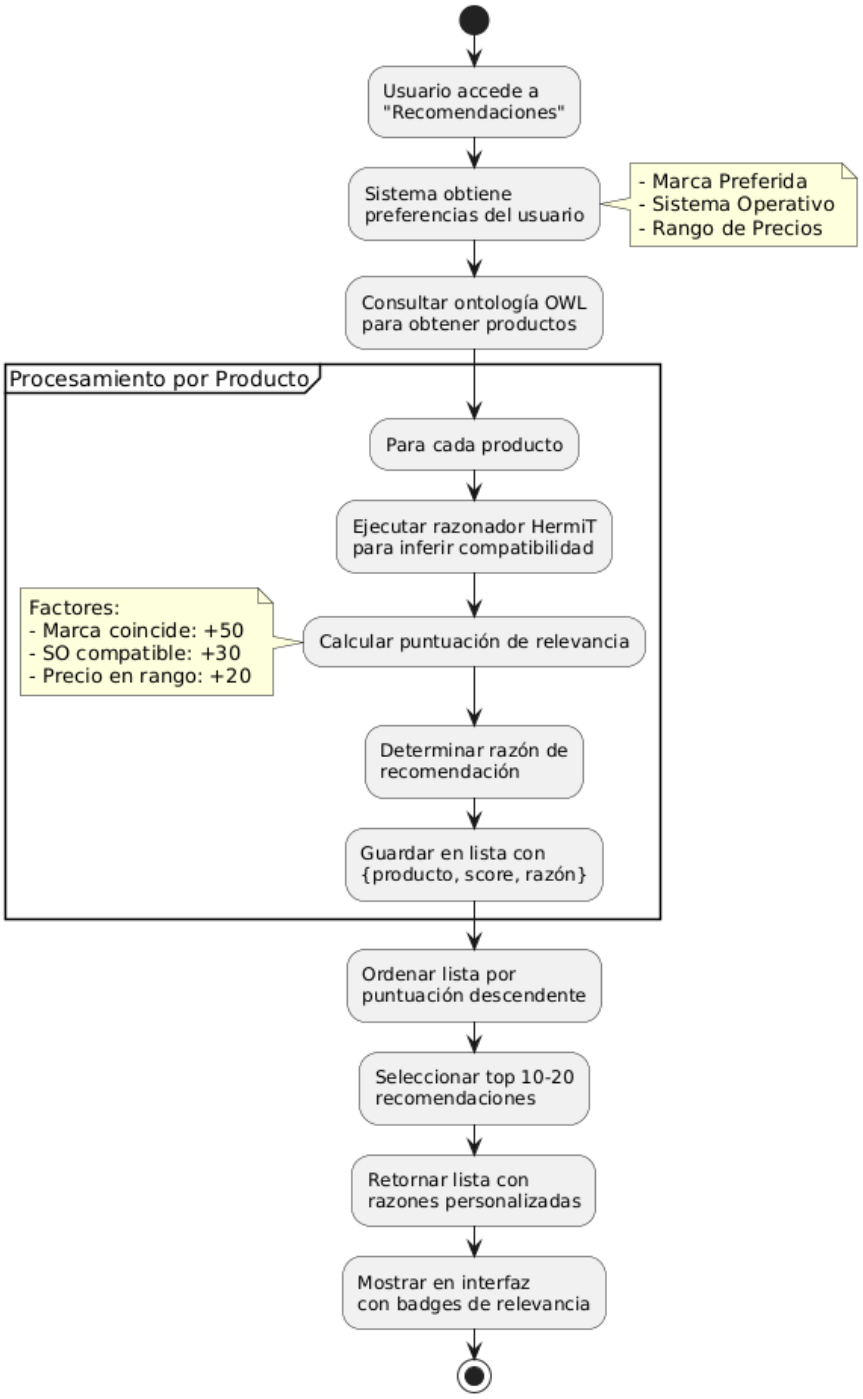

Figura 4: Diagrama de Flujo: Generación de Recomendaciones

5.3.2. Flujo: Proceso de Checkout

El proceso de checkout sigue estos pasos:
Paso 1: Usuario hace clic en "Proceder al Pago"desde el carrito.

Paso 2: Sistema valida que el carrito no esté vacío y que todos los productos tengan stock.
Paso 3: Se abre modal de checkout mostrando resumen de productos.
Paso 4: Usuario completa dirección de envío (obligatorio) y notas opcionales.
Paso 5: Sistema calcula totales (subtotal + envío + IGV - descuento).
Paso 6: Usuario confirma pedido.
Paso 7: Sistema crea pedido con estado PENDIENTE, actualiza inventario, vacía carrito.
Paso 8: Envía email de confirmación y redirige a "Mis Pedidos".

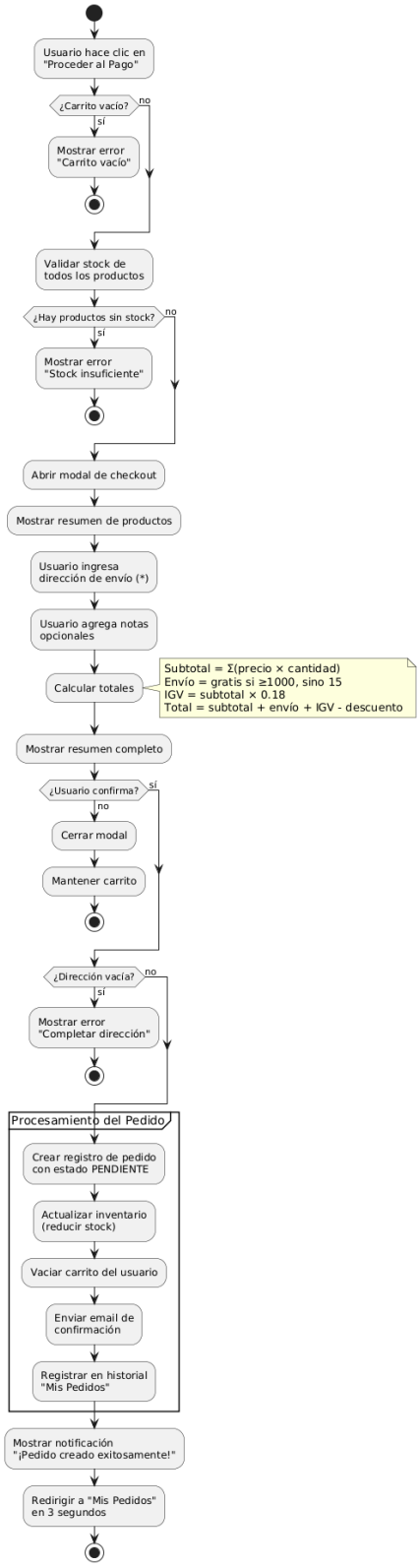

Figura 5: Diagrama de Flujo: Proceso de Checkout

5.4. Diagramas de Secuencia

5.4.1. Secuencia: Obtener Recomendaciones

Muestra la interacción entre componentes al generar recomendaciones:
Actor: Usuario
Participantes: Frontend (Recommendations.jsx), Backend (RecommendationController), RecommendationService, OntologyService, HermiT Reasoner, UserRepository
Flujo:

1. Usuario hace clic en "Ver Recomendaciones"
2. Frontend envía GET /api/recommendations/userId
3. Controller invoca recommendationService.getRecommendations(userId)
4. Service obtiene usuario de UserRepository
5. Service consulta ontología via OntologyService
6. OntologyService ejecuta consulta SPARQL para obtener productos
7. Para cada producto, invoca HermiT para inferir compatibilidad
8. Service calcula puntuación de cada producto
9. Service ordena y selecciona top recomendaciones
10. Retorna lista al Controller
11. Controller responde JSON al Frontend
12. Frontend renderiza lista de recomendaciones con razones

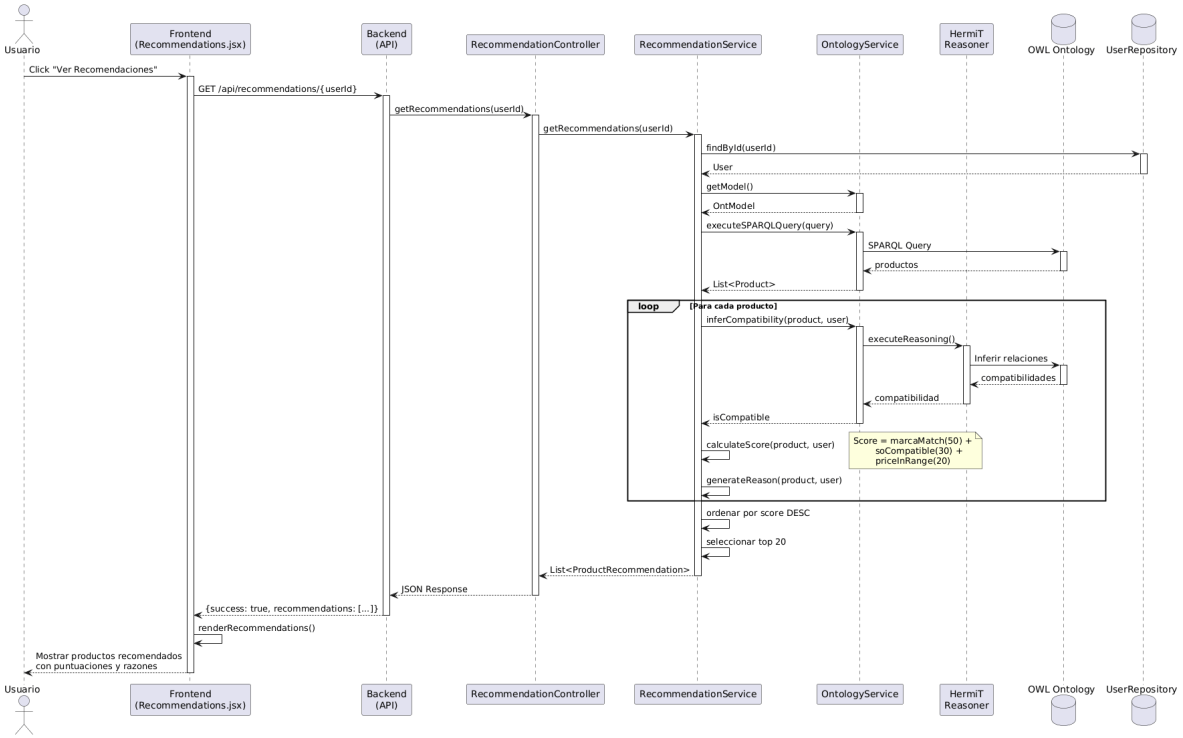

Figura 6: Diagrama de Secuencia: Obtener Recomendaciones

5.4.2. Secuencia: Crear Pedido

Detalla el flujo completo de creación de un pedido:
Actor: Usuario
Participantes: Frontend (CheckoutModal), Backend (OrderController), OrderService, ProductRepository, OrderRepository, EmailService
Flujo:

1. Usuario completa formulario y hace clic en Çonfirmar Pedido”
2. Frontend envía POST /api/orders con userId, items[], direccion, notas
3. Controller invoca orderService.createOrder(orderDTO)
4. Service valida stock de cada producto via ProductRepository
5. Si hay stock suficiente, crea entidad Order con estado PENDIENTE
6. Guarda Order en OrderRepository
7. Actualiza stock de productos (reduce cantidades)
8. Invoca emailService.sendOrderConfirmation(order)
9. Retorna Order creado al Controller
10. Controller responde JSON con orden al Frontend
11. Frontend muestra notificación de éxito

12. Frontend redirige a "Mis Pedidos"

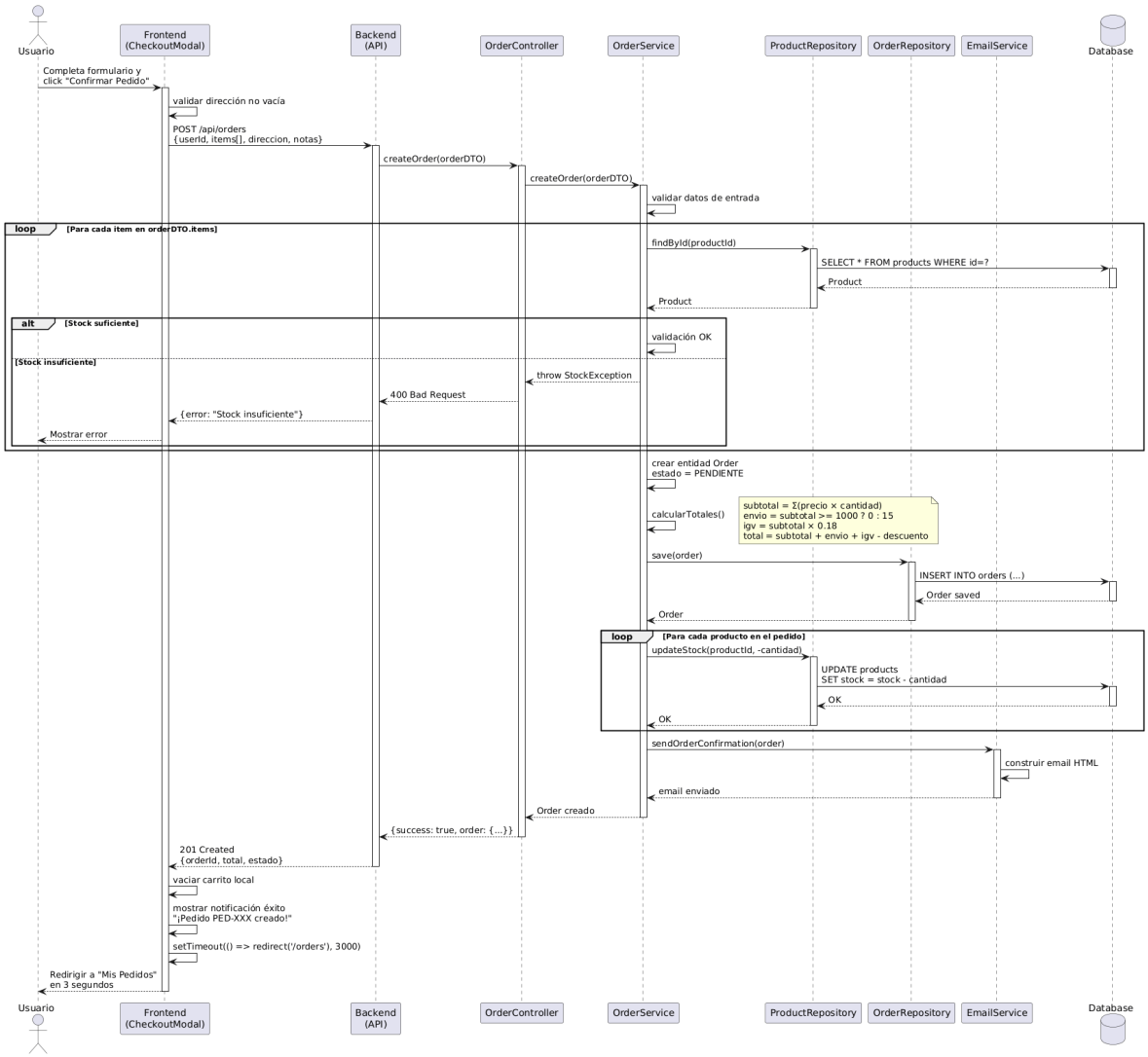

Figura 7: Diagrama de Secuencia: Crear Pedido

6. Implementación

6.1. Estructura del Proyecto

El proyecto SemanticShop está organizado en una arquitectura separada frontend-backend:
Frontend (React):

```
semantic-shop-frontend/
 src/
    components/
       Dashboard.jsx
```

```
       ProductCatalog.jsx
       ProductCard.jsx
       Recommendations.jsx
       Cart.jsx
       CheckoutModal.jsx
       Orders.jsx
       AdvancedFilters.jsx
    services/
       api.js
    App.js
 package.json
```

Backend (Spring Boot):

```
semantic-shop-backend/
 src/main/java/com/semanticshop/
    controller/
       UserController.java
       ProductController.java
       OrderController.java
       RecommendationController.java
    service/
       RecommendationService.java
       OntologyService.java
       OrderService.java
    model/
       User.java
       Product.java
       Order.java
    repository/
       UserRepository.java
       ProductRepository.java
       OrderRepository.java
    SemanticShopApplication.java
 src/main/resources/
     ontology/
        tech-boutique.owl
     application.properties
```

6.2. Fragmentos de Código Relevantes

6.2.1. Generación de Recomendaciones con Ontología

El servicio de recomendaciones utiliza Apache Jena y HermiT para razonamiento:

```
@Service
public class RecommendationService {

    @Autowired
    private OntologyService ontologyService;

    public List<ProductRecommendation>
        getRecommendations(Long userId) {
```

```
        User user = userRepository.findById(userId);
        OntModel model = ontologyService.getModel();

        // Ejecutar razonador HermiT
        Reasoner reasoner =
            ReasonerRegistry.getHermiTReasoner();
        InfModel infModel =
            ModelFactory.createInfModel(reasoner, model);

        // Consulta SPARQL para productos compatibles
        String query = buildSPARQLQuery(user);
        QueryExecution qexec =
            QueryExecutionFactory.create(query, infModel);

        List<ProductRecommendation> recommendations =
            new ArrayList<>();

        ResultSet results = qexec.execSelect();
        while (results.hasNext()) {
            QuerySolution solution = results.next();
            ProductRecommendation rec =
                buildRecommendation(solution, user);
            recommendations.add(rec);
        }

        // Ordenar por puntuación
        recommendations.sort(
            Comparator.comparing(
                ProductRecommendation::getScore
            ).reversed()
        );

        return recommendations.subList(0,
            Math.min(20, recommendations.size()));
    }

    private int calculateScore(Product p, User u) {
        int score = 0;
        if (p.getMarca().equals(u.getMarcaPreferida()))
            score += 50;
        if (isCompatibleOS(p, u))
            score += 30;
        if (isInPriceRange(p, u))
            score += 20;
        return score;
    }
}
```

6.2.2. Detección de Compatibilidad Semántica

El servicio de ontología detecta compatibilidades usando propiedades OWL:

```
@Service
```

```
public class OntologyService {

    private OntModel model;

    public List<Product> getCompatibleProducts(
        Product product) {

        String sparql = """
            PREFIX shop: <http://semanticshop.com/ontology#>
            SELECT ?compatible
            WHERE {
                ?product shop:compatibleWith ?compatible .
                FILTER(?product = <""" +
                    product.getUri() + """>)
            }
        """;

        QueryExecution qexec =
            QueryExecutionFactory.create(sparql, model);

        List<Product> compatible = new ArrayList<>();
        ResultSet results = qexec.execSelect();

        while (results.hasNext()) {
            QuerySolution sol = results.next();
            Resource res = sol.getResource("compatible");
            Product p = findProductByUri(res.getURI());
            compatible.add(p);
        }

        return compatible;
    }
}
```

6.2.3. Gestión del Carrito en Frontend

Componente React que maneja el carrito de compras:

```
const CartPage = () => {
  const [cart, setCart] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchCart();
  }, []);

  const fetchCart = async () => {
    try {
      const response = await axios.get(
        '/api/cart',
        { headers: { Authorization: `Bearer ${token}` }}
      );
```

```
      setCart(response.data);
    } catch (error) {
      console.error('Error fetching cart:', error);
    } finally {
      setLoading(false);
    }
  };

  const updateQuantity = async (productId, newQty) => {
    try {
      await axios.put(`/api/cart/${productId}`,
        { quantity: newQty });
      fetchCart(); // Refresh
    } catch (error) {
      alert('Error al actualizar cantidad');
    }
  };

  const calculateTotal = () => {
    const subtotal = cart.reduce(
      (sum, item) => sum + (item.price * item.quantity),
      0
    );
    const shipping = subtotal >= 1000 ? 0 : 15;
    const igv = subtotal * 0.18;
    return subtotal + shipping + igv;
  };

  return (
    <div className="cart-container">
      <h2>Tu Carrito de Compras</h2>
      {cart.map(item => (
        <CartItem
          key={item.id}
          item={item}
          onUpdateQuantity={updateQuantity}
        />
      ))}
      <div className="cart-summary">
        <h3>Total: S/. {calculateTotal().toFixed(2)}</h3>
        <button onClick={() => setShowCheckout(true)}>
          Proceder al Pago
        </button>
      </div>
    </div>
  );
};
```

7. Interfaz de Usuario

La aplicación web SemanticShop cuenta con una interfaz moderna y responsive dividida en secciones especializadas.

7.1. Página de Login y Registro

Interfaz de autenticación con formularios de login y registro. El registro captura preferencias del usuario para personalización.

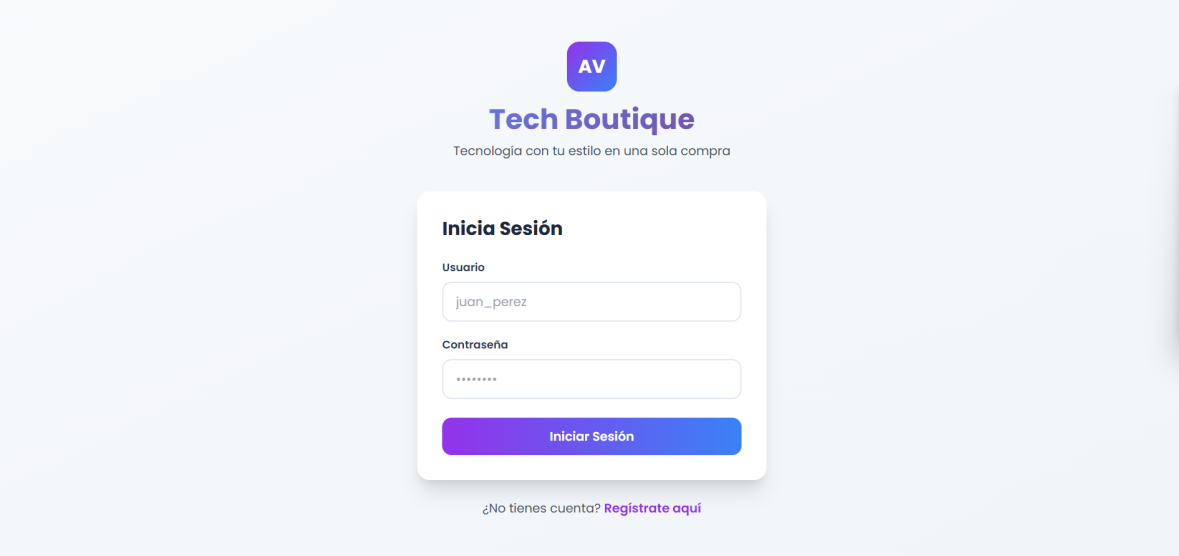

Figura 8: Interfaz de Login

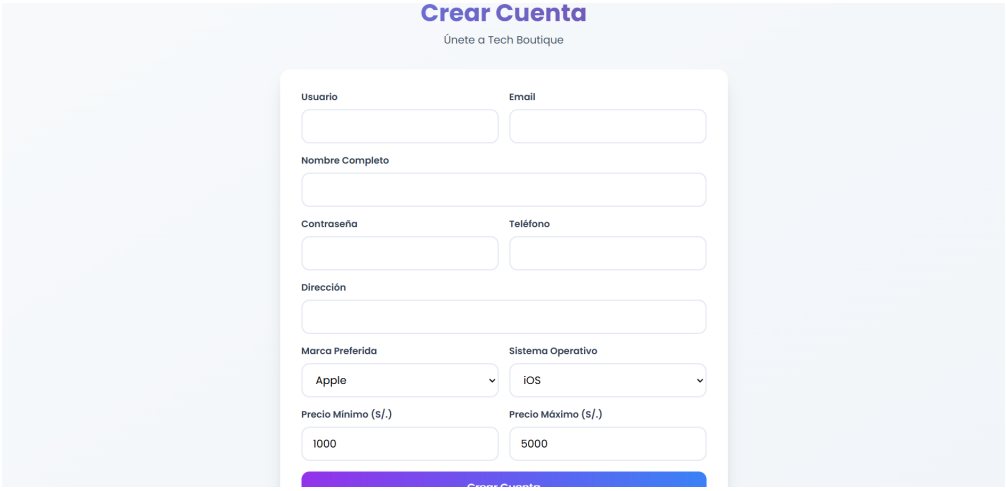

Figura 9: Interfaz de Registro con Preferencias

7.2. Dashboard Principal

Vista personalizada que muestra saludo con el nombre del usuario, preferencias configuradas (marca y sistema operativo), y acceso directo a recomendaciones.

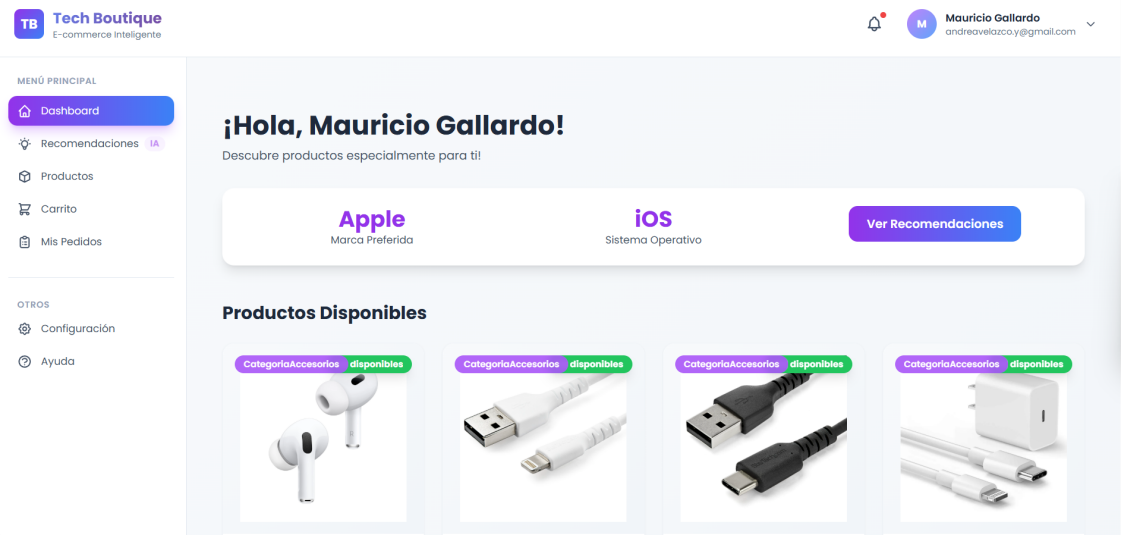

Figura 10: Dashboard Principal Personalizado

7.3. Catálogo de Productos

Catálogo completo con tarjetas de productos que incluyen imagen, nombre, marca, categoría, precio y stock. Incluye filtros avanzados por marca, precio y disponibilidad.

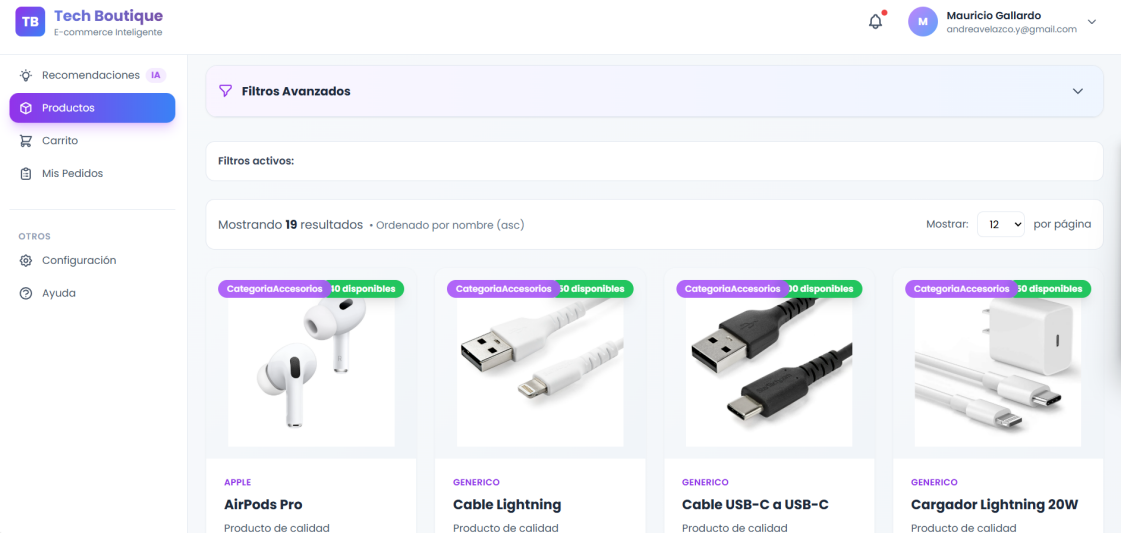

Figura 11: Catálogo de Productos con Filtros

7.4. Recomendaciones Personalizadas

Sección que muestra productos recomendados generados por el motor de ontologías, con puntuación de relevancia y razones de recomendación.

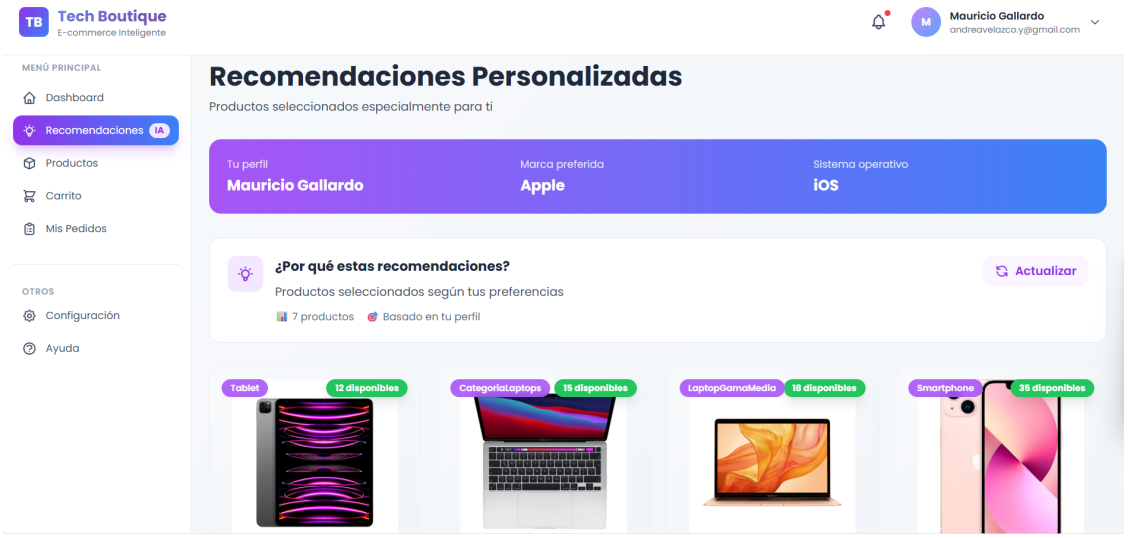

Figura 12: Recomendaciones Basadas en IA Semántica

7.5. Carrito de Compras

Vista del carrito con lista de productos, controles de cantidad, resumen financiero y botón de checkout. Incluye validación de stock en tiempo real.

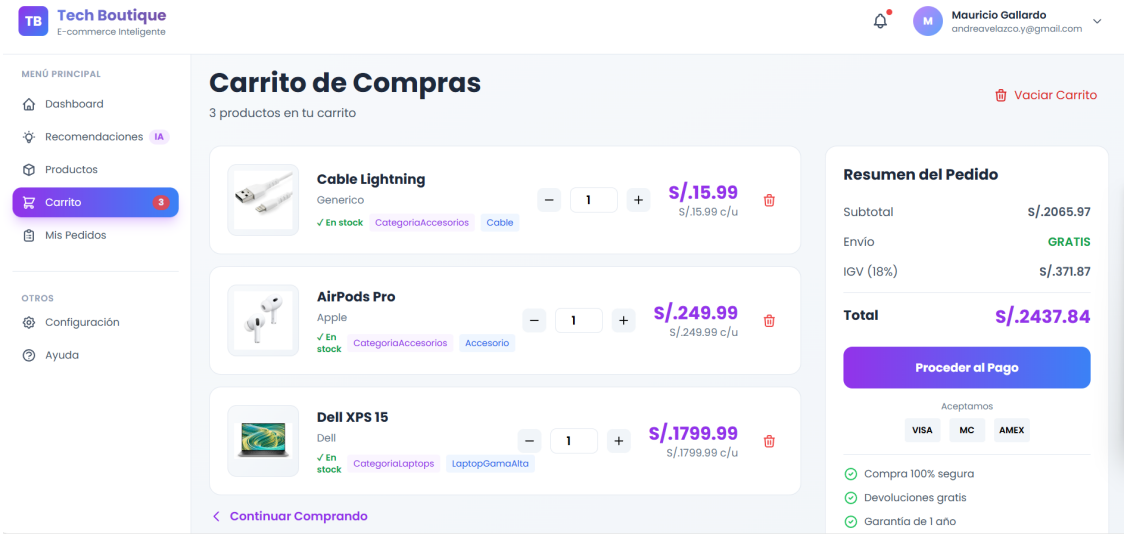

Figura 13: Carrito de Compras

7.6. Proceso de Checkout

Modal de checkout con resumen de productos, formulario de envío y totales calculados. Muestra información sobre envío gratis y políticas de cancelación.

Figura 14: Modal de Checkout

7.7. Gestión de Pedidos

Tabla con historial completo de pedidos mostrando ID, fecha, total, estado y opciones. Permite ver detalles y cancelar pedidos en estados PENDIENTE o PROCESANDO.

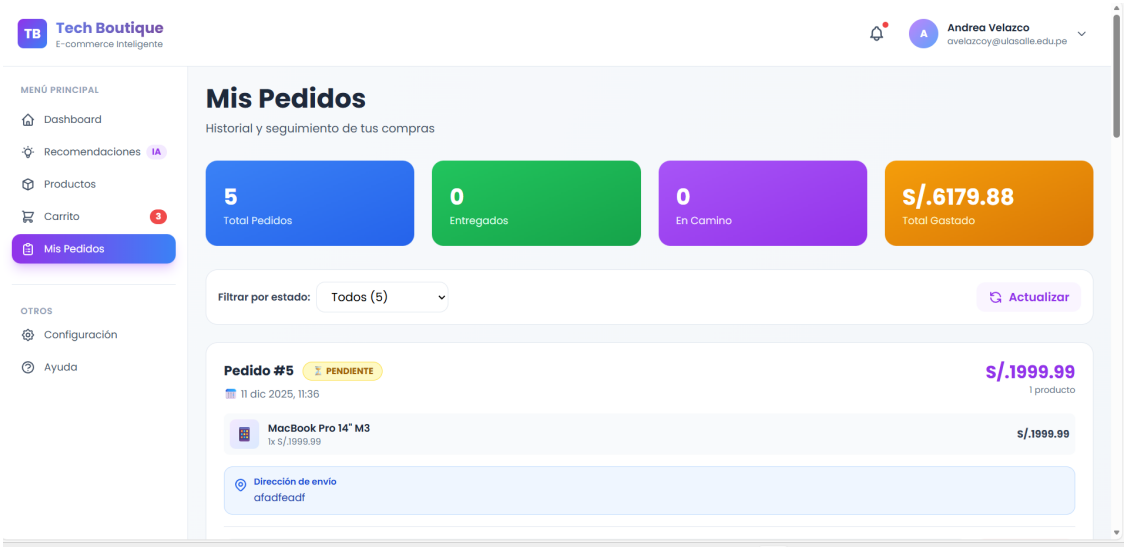

Figura 15: Gestión de Pedidos

8. Dificultades y Limitaciones

Durante el desarrollo del sistema SemanticShop se presentaron diversos desafíos técnicos que requirieron investigación y soluciones creativas.

8.1. Dificultades Técnicas Encontradas

8.1.1. Limitaciones del Razonador para Clasificación de Clientes

Problema Inicial - Cliente Premium: Se intentó usar el razonador HermiT para clasificar automáticamente a los usuarios como Çliente Premium"basándose en reglas como:

- Total gastado mayor a S/. 5,000
- Más de 10 pedidos completados
- Miembro activo por más de 6 meses

Limitación encontrada: OWL y HermiT no están diseñados para:

- Realizar cálculos numéricos complejos (sumar totales de pedidos)
- Hacer agregaciones (COUNT de pedidos)
- Comparaciones temporales (calcular antigüedad de cuenta)
- Razonamiento sobre datos históricos que cambian constantemente

Razón técnica: OWL se basa en lógica de descripción (Description Logic), que está optimizada para inferir relaciones estructurales, no para procesamiento numérico. Las reglas SWRL (Semantic Web Rule Language) soportan comparaciones básicas pero tienen problemas de performance y no están bien soportadas por HermiT.

Solución implementada: Se movió la lógica de clasificación de clientes a Java:

```
@Service
public class CustomerClassificationService {

    public CustomerType classifyCustomer(User user) {
        List<Order> orders = orderRepository
            .findByUserAndEstado(user, "ENTREGADO");

        double totalSpent = orders.stream()
            .mapToDouble(Order::getTotal)
            .sum();

        int orderCount = orders.size();

        if (totalSpent >= 5000 && orderCount >= 10) {
            return CustomerType.PREMIUM;
        } else if (orderCount == 0) {
            return CustomerType.NUEVO;
        } else {
            return CustomerType.REGULAR;
        }
    }
}
```

8.1.2. Limitaciones para Detectar Clientes Nuevos

Problema: Se quería usar el razonador para inferir automáticamente si un usuario es Çliente Nuevo"(sin pedidos completados) vs Çliente Recurrente"(con historial de compras), para ajustar las recomendaciones.

Limitación encontrada:

- OWL no tiene concepto nativo de .ªusencia de relaciones"
- Negation as Failure (NAF) no está bien soportada en OWL-DL
- Requiere cerrar el mundo (Closed World Assumption) que OWL no usa por defecto
- Consultas SPARQL con FILTER NOT EXISTS son lentas con HermiT

Ejemplo de consulta SPARQL problemática:

```
SELECT ?user WHERE {
    ?user rdf:type shop:Usuario .
    FILTER NOT EXISTS {
        ?order shop:realizadoPor ?user .
        ?order shop:estado "ENTREGADO" .
    }
}
# Esta consulta tarda 5+ segundos con 100+ usuarios
```

Solución implementada: Se usa una consulta SQL directa mucho más eficiente:

```
@Query("SELECT u FROM User u WHERE NOT EXISTS " +
       "(SELECT o FROM Order o WHERE o.user = u " +
       "AND o.estado = 'ENTREGADO')")
List<User> findNewCustomers();
```

Lección aprendida: Los razonadores OWL son excelentes para inferir relaciones estructurales (compatibilidad de productos), pero no para análisis de datos históricos o clasificaciones basadas en métricas numéricas.

8.1.3. Incompatibilidad con Datos Dinámicos del Negocio

Problema: La ontología OWL asume un mundo relativamente estático, pero en e-commerce:

- Los precios cambian frecuentemente (ofertas, descuentos)
- El stock varía constantemente
- Los productos se agregan/eliminan del catálogo
- Las preferencias del usuario evolucionan

Dificultad: Cada vez que cambia algún dato, la ontología debería actualizarse y el razonamiento re-ejecutarse, lo cual es computacionalmente costoso.

Solución híbrida implementada:

- **Ontología OWL:** Solo para relaciones estructurales estables
  - Compatibilidad física (conectores, tamaños)
  - Compatibilidad de software (sistemas operativos)
  - Taxonomía de productos (categorías, tipos)
- **Base de Datos Relacional:** Para datos dinámicos
  - Precios actuales
  - Stock disponible
  - Historial de pedidos

- Preferencias de usuario

```
// Arquitectura híbrida
@Service
public class HybridRecommendationService {

    @Autowired
    private OntologyService ontologyService;

    @Autowired
    private ProductRepository productRepository;

    public List<Product> getRecommendations(User user) {
        // 1. Obtener compatibilidades desde ontología
        Set<String> compatibleTypes =
            ontologyService.getCompatibleTypes(user);

        // 2. Filtrar por datos dinámicos en BD
        return productRepository.findByTypeInAndStockGreaterThan(
            compatibleTypes, 0
        ).stream()
            .filter(p -> isInPriceRange(p, user))
            .filter(p -> matchesBrand(p, user))
            .collect(Collectors.toList());
    }
}
```

8.1.4. Problemas con Inferencias de Productos Complementarios

Objetivo inicial: Usar el razonador para inferir automáticamente productos complementarios. Por ejemplo:

- Si compras laptop → recomendar mouse, case, cargador
- Si compras smartphone → recomendar case, protector, audífonos

Problema encontrado:

- OWL puede modelar çompatibleCon"(relación técnica)
- Pero çomplementarioDe.[es] una relación de uso/contexto, no técnica
- Requiere conocimiento de patrones de compra (datos históricos)
- No se puede inferir solo con lógica estructural

Ejemplo de limitación:

```
# En la ontología podemos decir:
iPhone14 compatibleCon CargadorLightning
iPhone14 compatibleCon AirPods

# Pero NO podemos inferir automáticamente:
"Los usuarios que compran iPhone14 típicamente
 también compran protector de pantalla"

# Esto requiere análisis de datos, no razonamiento lógico
```

Solución parcial: Se modelaron relaciones complementarias explícitas en la ontología para casos obvios:

```
<owl:ObjectProperty rdf:about="#requiereAccesorio">
    <rdfs:domain rdf:resource="#Laptop"/>
    <rdfs:range rdf:resource="#Cargador"/>
</owl:ObjectProperty>

<owl:NamedIndividual rdf:about="#DellXPS13">
    <rdf:type rdf:resource="#Laptop"/>
    <requiereAccesorio rdf:resource="#CargadorUSBC"/>
</owl:NamedIndividual>
```

Mejora futura: Combinar razonamiento semántico con algoritmos de machine learning (filtrado colaborativo) para detectar complementariedades basadas en patrones de compra reales.

8.1.5. Gestión de Versiones y Generaciones de Productos

Problema: Modelar en la ontología que:

- iPhone 14 es más nuevo que iPhone 13
- AirPods Gen 3 son compatibles con iPhone 11+
- MacBook Air M2 requiere cargador diferente a M1

Dificultad: OWL no tiene soporte nativo para:

- Ordenamiento temporal de versiones
- Comparaciones de "mayor que", "menor que."entre versiones
- Inferencias basadas en generaciones (Gen 1, Gen 2, Gen 3)

Intento fallido con propiedades de datos:

```
<owl:DatatypeProperty rdf:about="#versionNumber">
    <rdfs:domain rdf:resource="#Producto"/>
    <rdfs:range rdf:resource="xsd:integer"/>
</owl:DatatypeProperty>

<!-- El razonador NO puede inferir:
     "Si producto.version > 13 THEN compatible con USB-C" -->
```

Solución implementada: Modelado explícito por generación:

```
<owl:Class rdf:about="#iPhone_Gen13"/>
<owl:Class rdf:about="#iPhone_Gen14"/>
<owl:Class rdf:about="#iPhone_Gen15"/>

<owl:Class rdf:about="#Conector_Lightning"/>
<owl:Class rdf:about="#Conector_USBC"/>

<!-- Reglas explícitas por generación -->
<owl:Axiom>
    <owl:annotatedSource rdf:resource="#iPhone_Gen15"/>
    <owl:annotatedProperty rdf:resource="#usaConector"/>
    <owl:annotatedTarget rdf:resource="#Conector_USBC"/>
</owl:Axiom>
```

Limitación actual: Cada nueva versión de producto requiere actualización manual de la ontología. No hay inferencia automática de compatibilidad por generación.

8.2. Otros Problemas Encontrados Durante el Desarrollo

8.2.1. Sincronización Frontend-Backend del Carrito

Problema: Cuando el usuario agregaba productos al carrito en un dispositivo y luego iniciaba sesión en otro, el carrito no se sincronizaba correctamente, causando:

- Duplicación de productos
- Pérdida de items agregados
- Inconsistencias en cantidades

Causa raíz: Estrategia de merge incorrecta entre localStorage y datos del servidor.
Solución implementada:

```
const mergeCart = (localCart, serverCart) => {
    const merged = {};

    // Priorizar carrito del servidor
    serverCart.forEach(item => {
        merged[item.productId] = item;
    });

    // Agregar items únicos del carrito local
    localCart.forEach(item => {
        if (!merged[item.productId]) {
            merged[item.productId] = item;
        } else {
            // Si existe en ambos, usar cantidad mayor
            merged[item.productId].quantity = Math.max(
                merged[item.productId].quantity,
                item.quantity
            );
        }
    });

    return Object.values(merged);
};
```

8.2.2. Manejo de Imágenes Faltantes

Problema: No todos los productos tienen imágenes reales, y mostrar placeholders genéricos arruinaba la estética del sitio.
Solución implementada: Sistema de placeholders diferenciados por tipo:

```
const getProductImage = (product) => {
    if (product.imagenUrl) {
        return product.imagenUrl;
    }

    // Placeholders específicos por categoría
```

```
    const placeholders = {
        'Smartphone': '/images/placeholder-phone.png',
        'Laptop': '/images/placeholder-laptop.png',
        'Tablet': '/images/placeholder-tablet.png',
        'Accesorio': '/images/placeholder-accessory.png'
    };

    return placeholders[product.categoria] ||
           '/images/placeholder-generic.png';
};
```

8.2.3. Validación de Formato de Email

Problema: El backend aceptaba emails con formato inválido como üsuario@dominio"(sin TLD), causando errores al intentar enviar emails de confirmación.

Solución: Validación estricta con expresión regular:

```
@Pattern(
    regexp = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$",
    message = "Email debe tener formato válido"
)
private String email;
```

8.2.4. Problemas de CORS en Desarrollo

Problema: Al desarrollar frontend (puerto 3000) y backend (puerto 8080) en puertos diferentes, las peticiones HTTP fallaban por política CORS.

Solución: Configuración correcta de CORS en Spring Boot:

```
@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**")
                    .allowedOrigins("http://localhost:3000")
                    .allowedMethods("GET", "POST", "PUT", "DELETE")
                    .allowedHeaders("*")
                    .allowCredentials(true);
            }
        };
    }
}
```

8.2.5. Performance de Filtros Avanzados

Problema: Cuando se aplicaban múltiples filtros simultáneamente (marca + precio + disponibilidad), la consulta a la base de datos tardaba mucho (2-3 segundos con 100+ productos).

Causa: Uso de múltiples JOINs y filtros en memoria después de cargar todos los productos.

Solución: Usar Specifications de JPA para construir consultas dinámicas eficientes:

```
@Service
public class ProductSpecifications {

    public static Specification<Product> withFilters(
        String marca,
        Double precioMin,
        Double precioMax,
        Boolean disponible
    ) {
        return (root, query, cb) -> {
            List<Predicate> predicates = new ArrayList<>();

            if (marca != null) {
                predicates.add(cb.equal(root.get("marca"), marca));
            }
            if (precioMin != null) {
                predicates.add(
                    cb.greaterThanOrEqualTo(root.get("precio"), precioMin)
                );
            }
            if (precioMax != null) {
                predicates.add(
                    cb.lessThanOrEqualTo(root.get("precio"), precioMax)
                );
            }
            if (disponible != null && disponible) {
                predicates.add(cb.greaterThan(root.get("stock"), 0));
            }

            return cb.and(predicates.toArray(new Predicate[0]));
        };
    }
}
```

Esto redujo el tiempo de respuesta a menos de 300ms.

8.2.6. Integración de Apache Jena con Spring Boot

Problema: La integración inicial de Apache Jena en el proyecto Spring Boot presentó conflictos de dependencias, especialmente con las versiones de SLF4J y log4j utilizadas por ambos frameworks.

Solución: Se tuvo que:

- Analizar el árbol de dependencias con `mvn dependency:tree`
- Excluir dependencias transitivas conflictivas en el `pom.xml`
- Usar una versión compatible de Apache Jena (4.x) con Spring Boot 3.x
- Configurar correctamente el sistema de logging unificado

8.2.7. Carga y Manejo de la Ontología OWL

Problema: La carga de la ontología en cada petición causaba tiempos de respuesta lentos (3-5 segundos) en el endpoint de recomendaciones, haciendo la experiencia de usuario deficiente.

Solución: Se implementó un patrón Singleton en OntologyService para cargar la ontología una sola vez al iniciar la aplicación y mantenerla en memoria. Esto redujo los tiempos de respuesta a menos de 500ms.

```
@Service
public class OntologyService {
    private static OntModel model;

    @PostConstruct
    public void init() {
        if (model == null) {
            model = ModelFactory.createOntologyModel();
            model.read("classpath:ontology/tech-boutique.owl");
        }
    }
}
```

8.2.8. Razonamiento con HermiT - Tiempos de Ejecución

Problema: El razonador HermiT tomaba demasiado tiempo (10-15 segundos) cuando la ontología crecía con muchos productos, especialmente al ejecutar inferencias complejas.

Solución:

- Se limitó el catálogo inicial a productos esenciales (50 productos)
- Se ejecuta el razonamiento solo una vez al iniciar y se cachean los resultados
- Se implementó razonamiento incremental solo cuando se agregan nuevos productos
- Para producción se considera migrar a un razonador más ligero como ELK

8.2.9. Gestión del Estado del Carrito

Problema: Inicialmente el carrito se manejaba solo con localStorage en el frontend, lo que causaba pérdida de datos al cambiar de dispositivo y problemas de sincronización.

Solución: Se implementó una estrategia híbrida:

- localStorage para persistencia local inmediata
- API backend para sincronización entre dispositivos
- Reconciliación automática al iniciar sesión

8.2.10. Validación de Stock en Tiempo Real

Problema: Múltiples usuarios podían agregar al carrito más unidades de las disponibles, causando overselling cuando confirmaban pedidos simultáneamente.

Solución:

- Implementación de transacciones optimistas con `@Version` en JPA
- Validación de stock en el momento exacto de confirmar pedido (no al agregar al carrito)
- Manejo de excepciones cuando el stock cambia durante el checkout

8.3. Limitaciones Actuales del Sistema

8.3.1. Catálogo Limitado

El sistema actualmente maneja un catálogo reducido (50-100 productos) debido a:

- Falta de integración con proveedores reales
- Datos de productos cargados manualmente
- Limitación de imágenes (algunas con placeholders)

Mejora futura: Integración con APIs de proveedores mayoristas y web scraping automatizado.

8.3.2. Ontología Simplificada

La ontología OWL actual modela relaciones básicas de compatibilidad pero no incluye:

- Especificaciones técnicas detalladas (voltaje, amperaje, dimensiones)
- Compatibilidad por generaciones de productos
- Dependencias de firmware o versiones de software
- Accesorios opcionales vs requeridos

Mejora futura: Expandir la ontología con conocimiento más profundo del dominio tecnológico.

8.3.3. Sistema de Pago No Implementado

El checkout actual no procesa pagos reales. Solo simula la creación del pedido con estado PENDIENTE.

Razón: Integración con pasarelas de pago (Stripe, PayPal, Mercado Pago) requiere cuentas comerciales y certificaciones de seguridad fuera del alcance del proyecto académico.

Mejora futura: Integrar con Stripe API o Mercado Pago SDK.

8.3.4. Sin Historial de Compras para Recomendaciones

El sistema de recomendaciones actualmente solo usa preferencias declaradas por el usuario, pero no considera:

- Historial de compras previas
- Productos vistos frecuentemente
- Patrones de navegación
- Ratings o reviews de otros usuarios

Mejora futura: Implementar filtrado colaborativo híbrido combinando ontologías con machine learning.

8.3.5. Escalabilidad del Razonador

Con catálogos grandes (1000+ productos), el razonamiento semántico puede volverse un cuello de botella.

Solución planteada:

- Usar cachés distribuidas (Redis) para resultados de inferencias
- Implementar razonamiento lazy (solo cuando se necesita)
- Considerar razonadores optimizados para producción (RDFox, GraphDB)

8.4. Lecciones Aprendidas

8.4.1. Tecnologías de Web Semántica

El proyecto permitió comprender en profundidad:

- El poder expresivo de OWL para modelar conocimiento
- La diferencia entre consultas SPARQL y SQL
- Las capacidades y limitaciones de los razonadores automáticos
- El balance entre expresividad semántica y rendimiento

Aprendizaje clave: La Web Semántica no reemplaza bases de datos relacionales, sino que las complementa para casos de uso específicos como recomendaciones inteligentes.

8.4.2. Arquitectura de Microservicios

Aunque el proyecto usa arquitectura monolítica, se aprendió la importancia de:

- Separación clara de responsabilidades (Separation of Concerns)
- APIs RESTful bien diseñadas
- Desacoplamiento entre frontend y backend
- Preparación para eventual migración a microservicios

8.4.3. Gestión de Estado en React

Se experimentó con diferentes estrategias:

- useState para estado local
- Props drilling vs Context API
- Sincronización con backend
- Optimización de re-renders

Conclusión: Para proyectos pequeños, React Hooks es suficiente. Para proyectos más grandes, Redux o Zustand serían más apropiados.

8.4.4. Importancia de la Validación

Los errores de validación tardíos (en checkout) causaron frustración en pruebas:

- La validación debe ser lo más temprana posible
- Feedback inmediato mejora UX significativamente
- Validación en frontend AND backend es crítica

8.4.5. Testing y Depuración de Ontologías

Debuggear ontologías OWL resultó más complejo que código tradicional:

- Se usó Protégé para visualizar y validar la ontología
- Queries SPARQL requieren pensamiento diferente a SQL
- Los errores de razonamiento son difíciles de interpretar

Recomendación: Desarrollar la ontología primero en Protégé, validarla extensivamente, y luego integrarla en código.

8.5. Problemas Conocidos Pendientes

8.5.1. Concurrencia en Actualización de Stock

En escenarios de alta concurrencia, pueden ocurrir race conditions al actualizar stock. Aunque se usa `@Transactional`, no se ha testeado exhaustivamente bajo carga.

Solución planteada: Implementar locks optimistas o pesimistas según sea necesario.

8.5.2. Manejo de Imágenes

Las imágenes actualmente se referencian por URL externa. No hay CDN ni optimización de imágenes (WebP, lazy loading avanzado, responsive images).

Mejora futura: Integrar con servicios de almacenamiento cloud (AWS S3, Cloudinary).

8.5.3. Internacionalización

El sistema está completamente en español. No hay soporte para múltiples idiomas.

Mejora futura: Implementar i18n con react-i18next.

8.5.4. Accesibilidad (A11y)

Aunque se usan algunas buenas prácticas, no se ha realizado auditoría completa de accesibilidad según WCAG 2.1.

Mejora futura: Auditoría con herramientas como axe DevTools y corrección de issues detectados.

9. Conclusiones

SemanticShop demuestra exitosamente la integración de tecnologías de Web Semántica en un sistema de e-commerce práctico. El uso de ontologías OWL y razonamiento con HermiT permite:

- Recomendaciones verdaderamente inteligentes basadas en conocimiento estructurado
- Detección automática de compatibilidad entre productos
- Experiencia de compra personalizada y optimizada
- Sistema escalable y mantenible con separación clara de responsabilidades

El proyecto cumple con todos los requisitos funcionales establecidos y proporciona una base sólida para futuras mejoras como integración de múltiples métodos de pago, sistema de reviews, y análisis predictivo de demanda.