PROPUESTA DE UNA ARQUITECTURA DE BÚSQUEDA POR PATRONES Y GEOREFERENCIACIÓN

SEBASTIAN VELEZ RUIZ

UNIVERSIDAD EAFIT

ESCUELA DE INGENIERÍAS

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MEDELLÍN

2011

PROPUESTA DE UNA ARQUITECTURA DE BÚSQUEDA POR PATRONES Y GEOREFERENCIACIÓN

SEBASTIAN VELEZ RUIZ

Proyecto de grado para optar por el título de Ingeniero de Sistemas

Asesor

Jorge Hernán Abad Londoño

UNIVERSIDAD EAFIT

ESCUELA DE INGENIERÍAS

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MEDELLÍN

2011

Nota de aceptación

_____

_____

_____

_____

_____

_____

Firma del Presidente del Jurado

_____

Firma del Jurado

_____

Firma del Jurado

Medellín, 9 de Octubre de 2011

AGRADECIMIENTOS

En primer lugar me gustaría agradecer a mi asesor de tesis, Jorge Hernán Abad Londoño, por su apoyo incondicional durante un largo tramo de mi carrera. Gracias profesor por siempre responder positivamente en los momentos en que requería cambios, en los que tenía dudas frente al proyecto, y por su disponibilidad constante. Este proyecto no hubiera sido posible sin usted.

Además, me gustaría agradecer a 3 grandes amigos: Oscar Gómez, Vanessa Goez y Sergio Granada, quienes con su energía, esfuerzo y entusiasmo frente a sus proyectos, me dieron la inspiración y la motivación necesaria para sacar lo mejor de mí en todo momento.

Y además, la realización de este proyecto no habría sido posible sin mi familia, y no se habría podido disfrutar tanto de su realización sin la compañía de Juan Fernando Valencia, Carlos Hoyos y Alejandra Cruz, quienes hacen de cada día de trabajo un motivo para sonreír.

CONTENIDO

LISTA DE FIGURAS

GLOSARIO

Backend: Se refiere al componente del sistema que procesa peticiones desde la vista y entrega los resultados, sin tener ninguna interacción directa con los usuarios finales. Generalmente el backend se encarga de manejar la lógica del negocio así como las interacciones con la base de datos.

Bean: Componente Software que se puede reutilizar en varios lugares de la aplicación. En el proyecto de grado, los servicios se manejan como beans, de manera que se puedan inyectar con la ayuda de Spring en varios lugares de la aplicación.

CRUD: Acrónimo para Create/Read/Update/Delete, se usa comúnmente para referirse las operaciones básicas que se pueden realizar sobre una base de datos.

Dialecto: Dentro del léxico de Hibernate, un dialecto es el lenguaje particular de un motor de base de datos. Hibernate, por medio de dialectos, es capaz de convertir consultas en JPQL en consultas SQL genéricas.

Escalabilidad horizontal: Se refiere a la capacidad de agregar más nodos a un sistema de cualquier índole, tales como servidores, bases de datos, puertos, etc.

IDE: Siglas de Integrated Development Environment, es una herramienta que facilita el desarrollo de software por medio de un entorno empaquetado que permite, entre otras cosas, editar el código fuente, depurar, compilar, etc.

Inversion of Control (IoC): Práctica de ingeniería de software en la que el flujo de programación varía puesto que no se hacen llamados procedimentales a métodos, sino que estos son llamados por una entidad coordinadora.

ORM: Estrategia de desarrollo que permite tomar provecho de la orientación a objetos en las bases de datos. Permite relacionar clases con tablas o vistas, y sus atributos con columnas.

Runtime: Se refiere al momento en que una aplicación se encuentra en ejecución. Generalmente se usa este término para definir las acciones que deben ocurrir mientras la aplicación se activa, sin involucrar ningún tipo de cambios que requieran compilar de nuevo el código fuente.

Spam: Envío de mensajes no solicitados por diversos medios. El más conocido es spam de correos electrónicos, pero puede aplicar para multitud de medios.

Stand alone: Tipo de aplicación que corre sin necesidad de conexión a ninguna red, por medio de una plataforma cliente rico que se encuentra instalado en un computador.

0. INTRODUCCIÓN

La evolución de la ingeniería de software y el “boom” de Internet ha permitido la creación de nuevos modelos de negocios virtuales, nuevas formas de comunicación y de cierta manera, ha cambiado la forma en que las personas interactúan con el ambiente que los rodea. Más recientemente, se ha dado el ingreso de la georeferenciación y de la pregunta “¿dónde estoy?”, como se puede ver en aplicaciones que han tenido una gran acogida como las redes sociales Foursquare, y los mismos Facebook y Twitter, que tienen este tipo de funcionalidades.

Sin embargo, cuando se va al terreno de las búsquedas de establecimientos en Colombia, la localización de lugares sigue teniendo un modelo de negocio muy estático, es decir, las búsquedas se realizan sobre algunas palabras claves previamente ingresadas, y los resultados se determinan por las coincidencias exactas de estas palabras.

Es posible, haciendo uso de diversas estrategias, crear un modelo de búsqueda de establecimientos escalable, intuitiva y mucho más personalizada, que involucre el concepto de la ubicación actual de la persona, y que permita encontrar resultados más relevantes y afines a las motivaciones del usuario. De esta manera, las búsquedas sobre porciones exactas de texto quedarían relegadas a ciertas situaciones particulares, y el potencial del manejo de la semántica de la oración se podría explotar en una gran medida.

1. OBJETIVOS

1.1 OBJETIVO GENERAL

Diseñar, proponer e implementar una arquitectura de búsqueda de lugares basada en una serie de patrones del lenguaje español, y de manejo de georeferenciación, que permita encontrar lugares de una manera más intuitiva y cercana al ser humano que las búsquedas de texto tradicionales.

1.2 OBJETIVOS ESPECÍFICOS

- Identificar un conjunto de tecnologías / frameworks que permitan desarrollar la solución de manera efectiva.
- Analizar, diseñar e implementar un modelo que permita la escalabilidad y la eficiencia de la aplicación.
- Identificar patrones lingüísticos que permitan crear consultas de acuerdo a expresiones que un usuario tenga, incluyendo palabras coloquiales o populares (jergas).
- Permitir el manejo de múltiples ciudades dentro de la aplicación, de manera que todas las búsquedas se filtren para una determinada porción de los lugares registrados.
- Combinar el proceso de búsqueda con unas coordenadas, opcionales, que defina el usuario, de manera que la georeferenciación tome un lugar importante en la aplicación.

- Implementar el algoritmo y crear una interfaz simple de consulta que lo utilice.
- Crear diagramas de arquitectura, modelo de dominio, de secuencias, o aquellos a los que de lugar.

2. TRABAJOS SIMILARES – MARCO TEÓRICO

El proyecto de grado se encuentra enmarcado en el campo del procesamiento del lenguaje natural (con siglas NLP – Natural Language Processing). Este campo de la inteligencia artificial tiene como objetivo la comprensión de lenguaje humano de una manera intuitiva y cómoda para el usuario, y de esta manera facilitar la comunicación usuario – máquina y mejorar la experiencia final.

Históricamente, dentro de los mayores problemas del NLP se encuentran la ambigüedad del lenguaje, y el manejo de sinónimos. Además, para llegar a ser realmente intuitivos se debe tener en cuenta demasiadas variantes producto de las diferencias del idioma entre regiones, expresiones coloquiales, etc.

Dentro del NLP se encuentran múltiples variantes y estrategias. Una división popular es la dada entre la aproximación estadística y la aproximación lingüística:

- Aproximación estadística: Manejo de palabras claves en los posibles resultados de una búsqueda. Generalmente consiste en un mapeo entre palabras del resultado y palabras de la búsqueda. Adicionalmente se puede asignar un peso a los mapeos, buscando determinar relevancia.
- Aproximación lingüística: Permite el análisis de expresiones por medio de un reconocimiento de su estructura tanto morfológica, sintáctica y semántica. La idea de este modelo es tratar de entender las motivaciones y estructuras del idioma de quien ejecuta la consulta, y a partir de allí generar una serie de resultados de acuerdo a esa interpretación que se realizó sobre la palabra original.

2 ejemplos de sistemas que se fundamentan o que usan este tipo de aproximaciones son:

START – Natural Languague Question Answering System: Iniciado por Boriz Katz en el MIT desde 1993, y en permanente desarrollo, es un sistema de preguntas y respuestas sobre múltiples temas, basado en estrategias avanzadas de análisis sintáctico. Se puede encontrar en el sitio web http://start.csail.mit.edu/.

Wolfram Alpha: Desarrollado por Wolfram Research, sistema que mantiene una base de datos de conocimiento de gran magnitud, y que interpreta las consultas que se le realizan para tomar ciertas porciones de datos de esta base de conocimiento. Maneja una gran cantidad de áreas del saber.

Una sub rama de la NLP, sobre la que se enfoca este proyecto de grado, es sobre el reconocimiento de patrones. Esta técnica se puede entender fácilmente si se considera el manejo que tienen la mayoría de proveedores de correo electrónico modernos del spam. Generalmente, con base a ciertos textos en el cuerpo del correo (publicidad, expresiones cautivadoras, etc.) se decide cuando un texto es probablemente un spam. En general, trata de analizar expresiones, frases, o simples palabras, y a partir de allí tomar ciertas decisiones.

Particularmente para el idioma español, existen unas desventajas adicionales, como el hecho de no contar con una estructura fija para los tiempos verbales, e incluso con inconsistencias en el mismo tiempo verbal pero para diferentes pronombres.

3. PROPUESTA DE UNA ARQUITECTURA DE BÚSQUEDA POR PATRONES Y GEOREFERENCIACIÓN

3.1. Ontología del dominio

Una ontología se puede definir como la realización de un esquema conceptual que permite formalizar un dominio determinado. A través de su realización es posible hacer explícito el alcance del proyecto, determinando las características de las expresiones que se desean soportar.

Para el desarrollo de la ontología se tuvo como referencia la guía propuesta por Natalia F. Noy y Deborah L. McGuinness de la universidad de Stanford. El objetivo de este ejercicio es crear el modelo de dominio de las búsquedas, lo que permitirá definir los tipos de expresiones a soportar, sus características, y a partir de esto se podrá crear un esquema objetual dentro del proyecto.

Paso 1: Dominio y alcance de la ontología

La ontología cubre el dominio de las expresiones que se usan al momento de realizar búsquedas sobre establecimientos comerciales de una ciudad. Se realiza la ontología buscando encontrar categorías o patrones en las expresiones que se pueden realizar, y hacer explícito los tipos de oraciones que se soportan en el proyecto de grado.
La ontología debe permitir definir tipos de términos dentro de una oración que proveen información significativa para encontrar un establecimiento. Contempla tanto expresiones que permitan filtrar los establecimientos por criterios dados, como adjetivos que permiten compararlos.

Como ejemplos de oraciones que la ontología debe soportar se encuentran:

- Restaurantes de comida italiana
- Tiendas de camisetas cerca de Sabaneta
- Bares con cocktails y que coloquen música tipo Rock
- Restaurantes abiertos los sábados de 8am a 3pm en Envigado
- Tiendas baratas de zapatos
- Restaurantes nuevos
- Restaurantes con platos de menos de 30000
- Sitios en Laureles

Paso 2: Enumeración de términos importantes para la ontología

La siguiente es una lista de términos importantes para la ontología:

- Lugar
- Precio
- Ubicación
- Comparaciones
- Caro
- Exclusivo
- Horario
- Abierto
- Cerrado
- Horas
- Días
- Barato
- Nuevo

- Barrio
- Coordenada
- Tipos de comida
- Sector
- Restaurante
- Bar
- Tienda
- Artículos
- Tipo de licor
- Menor que
- Mayor que
- Tipo de lugar
- Cerca

Paso 3: Definir las clases y subclases

Usando un enfoque top-bottom, se organizan todos los conceptos previamente mencionados en clases y subclases (No estamos hablando de clase en el concepto de programación orientada a objetos). Reorganizando los conceptos según agrupaciones lógicas se tiene:

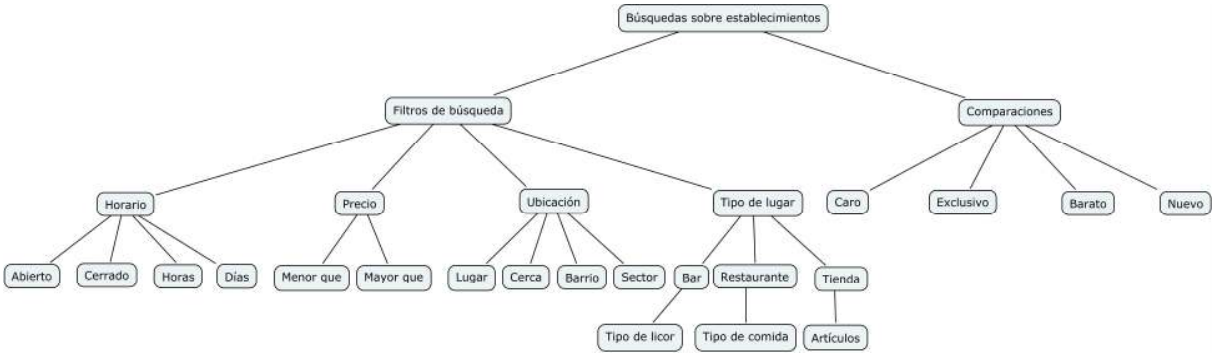

Ilustración 1: Ontología del domino - Conceptos relevantes

Paso 4: Propiedades de las clases – Slots

De este enfoque top-bottom se logran identificar ciertas particularidades que son importantes para mencionar:

- Las búsquedas se pueden realizar sobre los horarios de un lugar (Sobre días particulares, horas, momentos en que se encuentra cerrado o abierto).
- También pueden existir búsquedas sobre los precios de los productos que ofrece un establecimiento, filtrando por un precio promedio que se debe identificar.
- Adicionalmente, es posible georeferenciar los lugares que se desean encontrar. Es posible realizarlo brindando unas coordenadas precisas, o filtrando los resultados por lo cerca que se encuentren a un determinado sector, como un barrio.
- Por último, es posible filtrar por los diferentes tipos de lugares que ofrece la aplicación. Para el caso de este proyecto se ejemplifica con bares, restaurantes y tiendas, por lo que éstos son incluidos en la ontología. Sin embargo, la idea es mantener una aplicación abierta a cualquier dominio. Para cada tipo de establecimiento es posible indicar atributos particulares.
- Además, otra categoría son las comparaciones, que permiten dar una noción de orden a los establecimientos con expresiones como caro, barato, nuevo, etc.

Los slots se pueden definir como las propiedades particulares de las clases creadas. Algunas clases del primer modelo ya contienen los slots (como es el caso de los tipos de lugares).

Dada la naturaleza de la ontología, no se tienen muchos Slots. La ontología no gobierna sobre las propiedades de los sitios, sino sobre las formas de buscar sobre ellos. Los tipos de lugares los contienen, pero en las otras ramas se pueden observar instancias particulares, o ejemplos, de estas categorías de búsquedas ya mencionadas.

Paso 5: Facetas de los Slots

Todos los slots modelados son definidos como Strings. Su cardinalidad es 1 a 1, puesto que cada elemento del buscador responde a una característica particular del modelamiento del buscador. Es posible, obviamente, que dentro de una misma búsqueda se encuentren múltiples categorías de resultados.

Los siguientes pasos brindan un refinamiento de los términos ya expuestos. El diagrama presentado en el paso 3 corresponde con la última versión de la ontología generada. A continuación se presentan algunas particularidades de ésta:

- Es posible crear más instancias (particularidades dentro de los tipos de establecimientos), por lo que el buscador debe soportar tantos atributos como se definan a los tipos de establecimientos.
- Se pueden crear entonces 5 características funcionales del buscador: Comparadores, búsquedas de horario, búsquedas basadas en precio, basadas en tipos de lugares y basadas en la ubicación. Cada una de estas, en una etapa posterior del proyecto, se convertirán en los patrones usados en la aplicación.

- El alcance del buscador puede crecer tanto en instancias, en ejemplos de una categoría de búsqueda, o en una categoría de alto nivel. Se debe mantener al máximo la flexibilidad para crecer en cualquier rama.

A continuación se verá la transición desde este esquema conceptual hacia un esquema lógico que culminará con la implementación del buscador dentro del sistema.

3.2. Introducción a la solución

A través de un conjunto amplio de oraciones que se espera soportar con el algoritmo, se logran identificar 2 familias de expresiones lingüísticas que serán sobre los que tratará este proyecto de grado. A grandes rasgos, estas expresiones son las siguientes:

Adjetivos: Dentro del buscador, un adjetivo es una forma de calificar un resultado. Es la única forma de valorar, o comparar, 2 determinados lugares. Ejemplos de adjetivos son palabras como: barato, caro, lindo, nuevo, viejo, antiguo, etc. Para sintetizar, es cualquier expresión que pueda generar una comparación y crear un orden.

Restricciones: Diferentes tipos de expresiones que limitan o filtran un rango de búsqueda. Las restricciones son condiciones que deben cumplirse por parte de todos los elementos a retornar en una búsqueda. Esta categoría es muy amplia, y a lo largo del proyecto se verá en detalle cada una de las posibilidades abarcadas dentro del buscador. Ejemplos de restricciones son expresiones como: “quiero solo restaurantes”, “lugares de comida italiana”, “lugares cerca a mi casa”, etc. No es posible comparar 2 lugares por medio de restricciones.

El proyecto de grado crea un modelo de dominio de diferentes tipos de lugares (aunque se tiene en cuenta atributos particulares de unos cuantos tipos de lugares, el modelo es lo suficientemente genérico como para adaptarlo a más tipos), y por medio de las expresiones expuestas logra transformar oraciones completas en consultas a una base de datos (por medio de un ORM) con el fin de encontrar lugares de una manera más intuitiva y buscando intuir las intenciones de búsqueda del usuario. Los resultados, en las pruebas realizadas, han sido muy satisfactorias, y se ha logrado un nivel de precisión alto.

Al momento de diseñar el buscador, se tuvieron en cuenta tres cualidades fundamentales. Todo el desarrollo se realizó teniendo en la mira estos 3 atributos:

Eficiencia: Dado que la cantidad de patrones que contiene el sistema puede llegar a ser alto, se mantuvo en la mira durante todo el desarrollo del proyecto que el proceso de búsqueda tuviera una alta eficiencia, evitando toda redundancia, manteniendo la complejidad algorítmica en el nivel más bajo posible. Como se verá en detalle, generalmente se utilizaron estructuras de datos que almacenan la información y los patrones en árboles, lo que permite un procesamiento eficiente.

Escalabilidad: La aplicación se pensó de tal manera que sea capaz de soportar gran cantidad de datos. Como se verá más adelante, se crearon diversos modelos que permiten mantener un rendimiento estable de la aplicación ante altos volúmenes de información.

Extensibilidad: La posibilidad de incluir nuevos soporte para nuevos tipos de expresiones en el futuro, así como incluir nuevas posibilidades dentro de las expresiones previamente incluidas, siempre fue contemplada.

Dado el gran número de patrones que se pueden crear, todos estos se encuentran en una base de datos. Esto permite que la aplicación crezca y, a medida que se identifiquen nuevos patrones, se puedan ingresar al sistema.

3.3 Estructura general de la solución

La aplicación consiste en un backend que carga un conjunto de patrones almacenados en base de datos. Estos patrones son almacenados en memoria, generalmente en estructuras de árboles. Cada patrón tiene la capacidad de decidir cuando un texto cumple o no con sus criterios. La aplicación busca, para un texto que se ingresa, los patrones que coinciden con un determinado texto, totalmente en memoria. Los patrones que aplican para el texto son recopilados, agrupados, y a partir de ellos se genera una sentencia JPQL que consigue los sitios que cumplen con los criterios del usuario.

La aplicación, además, está conformada por una serie de servicio s que permiten realizar algunas operaciones CRUD del sistema. El acceso a datos no es solo para cargar los patrones, sino para almacenar información de ciudades, establecimientos, ubicaciones de los establecimientos, barrios o sectores, etc. Por cada entidad de negocio se ofrecen unos servicios que permiten la gestión de ella.

Los patrones determinan la consulta, sin embargo, ésta debe hacerse en una base de datos. Para lograr escalabilidad horizontal, dado que la aplicación debe responder de una manera eficiente frente a grandes volúmenes de datos, se han creado diferentes bases de datos, una por cada ciudad, y se ha utilizado la herramienta Hibernate de Jboss para brindar transparencia en la codificación frente a la fuente de datos que se usa.

La aplicación funciona correctamente como una aplicación stand alone o como parte de una aplicación web. Todas las consultas se deben aplicar sobre una ciudad, pues no tiene

sentido que un usuario quiera mezclar información entre diferentes ciudades. Para esto, cada hilo que ejecute la aplicación trae consigo a modo de "variable" un código de ciudad. El código de ciudad se puede obtener en toda transacción, buscando saber la fuente de datos particular para esta ciudad. Desde el punto de vista de desarrollo, en la mayoría de las operaciones es transparente este proceso, pues se encapsularon las operaciones de obtención de conexiones y ejecución de consultas en unos pocos artefactos que usa toda operación específica del sistema.

El caso de una aplicación web está fuera del alcance en implementación, sin embargo, se debe mencionar que la aplicación se adapta perfectamente a este tipo de desarrollos, asociando cada hilo del sistema a una ciudad, por medio de una cookie o de una variable en sesión.

3.4 Herramientas seleccionadas

A continuación se enumeran las tecnologías usadas para la solución del proyecto de grado, y una breve descripción de cada una:

Java: La plataforma elegida para la solución ha sido Java. Las razones que han llevado a elegirla son las siguientes:

- Existen numerosas librerías gratuitas para la mayoría de los componentes que se requerían dentro del marco del proyecto.
- Java es altamente extensible.
- Integración con frameworks web.
- Soporte para desarrollo orientado a componentes e inversión de control.
- Alta documentación en línea.
- Aceptación en el mercado.

- Posibilidad de obtener código fuente de librerías de terceros, en caso que se requiera modificar el comportamiento de alguna de ella.

En realidad Java permite numerosos estilos de programación, dependiendo de las herramientas sobre la que se trabaje. Por lo que se dará mayor énfasis a éstas.

Hibernate: Framework de persistencia que permite relacionar clases Java con tablas de base de datos por medio de anotaciones en las clases o de archivos XML. La idea principal de Hibernate es no perder las ventajas que ofrece la programación orientada a objetos cuando se habla de bases de datos, y por medio de su ORM (Object – Relational Mapping) logra crear consultas de base de datos que hablan sobre objetos Java, donde se mapean las columnas de base de datos con algunos atributos de la clase.
Hibernate cuenta con una implementación propia de ORM, cuyo lenguaje de consulta es HQL, y también se apega al estándar JPA, que se verá a continuación. Para este proyecto de grado se usa la implementación de JPA de Hibernate.
Razones para usar Hibernate:

- Independencia de base de datos.
- Permite utilizar el modelo de dominio como modelo de base de datos.
- Facilidad para generar un modelo de multitenancy.
- Esquemas de herencia y relaciones entre entidades.
- Alta documentación e integración con otras plataformas.

JPA: API de persistencia de Java que consiste en 3 elementos:

- Un conjunto de clases y de interfaces que definen las firmas y las capacidades de la aplicación.
- Estándar para realizar mapeos objeto relacionales.

- Lenguaje de consulta (JPQL) en el cual se realizan consultas sobre los objetos y atributos, y a su vez, también se retornan objetos, independizándolos del modelo de base de datos.

Se usa JPA por su facilidad de acoplamiento con las necesidades de herencia, multitenancy y extensibilidad de la aplicación, además, por que es un estándar altamente reconocido en el medio y con alto nivel de documentación.

Spring: Framework Java que se ha convertido en los últimos años en una alternativa del estándar de Enterprise Java Beans. Spring acumula varias de las mejores prácticas de programación Java del momento. Para el proyecto de grado, se han usado las siguientes funcionalidades de Spring:

- Inversión de control: Popularmente conocido como IoC, busca mantener un bajo acoplamiento entre los componentes de un sistema, por medio del manejo de interfaces y de la inyección de dependencias entre componentes, es decir, evitando la inicialización directa entre entidades de negocio.
- Transaccionalidad: Spring maneja por medio de anotaciones y de programación orientada a aspectos la transaccionalidad de base de datos.

Junit: Framework que permite la ejecución de pruebas unitarias dentro de la plataforma Java. El desarrollo del proyecto y sus resultados utilizaron la herramienta de pruebas unitarias Junit, entre otros, por los siguientes motivos:

- Permite verificar automáticamente los resultados que arroja el sistema.
- Alta integración con Spring.
- Facilidad de manejar pruebas de regresión, es decir, verificar que añadiendo nuevas funcionalidades al sistema se alteré el correcto funcionamiento de un componente ya finalizado.

- Facilidad de integración con el IDE usado (eclipse)

3.5 Arquitectura del sistema

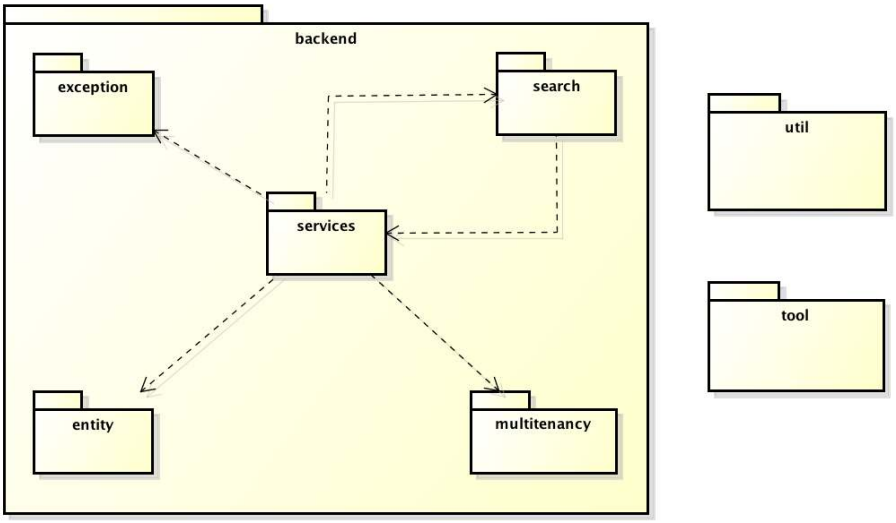

Ilustración 2: diagrama de paquetes de la aplicación

La estructura básica del buscador es un backend que consiste en 5 paquetes secundarios:

Entity: Paquete que contiene todas las entidades de negocio del sistema. Una entidad de negocio es un objeto que representa un sustantivo del domino del sistema. Las clases que representan entidades heredan de la clase BaseEntity. Hibernate se encarga de crear la base de datos automáticamente, a partir del modelo de dominio especificado en este paquete. Las clases no persistibles no se encuentran aquí.

Multitenancy: El paquete multitenancy consiste en las clases que facilitan el manejo de diferentes bases de datos en el sistema. Las responsabilidades de las clases de este paquete son las siguientes:

- Proveer las conexiones a base de datos necesarias para toda operación del sistema, teniendo en cuenta la ciudad asociada al hilo.
- Garantizar la transaccionalidad de las operaciones. (Apoyándose en Spring para esto)
- Servir de base para los servicios, buscando que estos tengan un manejo transparente de las conexiones a la base de datos. Para esto, este paquete crea ciertas clases bases para los servicios que facilitan el manejo del acceso a datos.

Exception: Paquete que contiene las diferentes excepciones personalizadas que tiene el sistema.

Services: Paquete principal de la aplicación en conjunto con search. Este paquete contiene todos los servicios que cada entidad ofrece de manera pública. La estructura de sub-paquetes, de interfaces, y de implementaciones de estas mapean con la estructura de los paquetes y nombres de las entidades, es decir, que por cada entidad se crea un servicio que permite manejar las operaciones CRUD de ésta entidad.

Para el acceso a servicios, buscando desacoplar las interfaces de las implementaciones, se utiliza Spring como framework que facilite una práctica de ingeniería de software llamada inversión de control. La inversión de control permite que las clases que utilicen un determinado servicio no tengan una dependencia directa con la implementación, sino que simplemente utilicen la interfaz, que sirve a modo de contrato como una definición de los servicios de la clase. En caso que se requiera cambiar la implementación, la forma en que se usa un servicio, ninguno de los componentes que lo usan se ve afectado, y esto permite un desarrollo orientado a componentes independientes.

Para la transaccionalidad y el acceso a datos, la capa de servicios utiliza las clases del paquete multitenancy. Para un servicio, es prácticamente transparente el “dónde” se almacena o se recupera un dato. Esto se explicará con más detalle en otras secciones.

Search: Paquete que encapsula la lógica de la búsqueda por patrones. En este paquete se encuentra la lógica con que se construyen las consultas en bases de datos a partir de una localización de patrones. Sin embargo, vale la pena aclarar que la mayoría de los patrones se encuentran en base de datos, por lo que se consideran entidades persistibles y se ubican en el paquete entity. En este paquete se encuentra todo el proceso de detección de patrones y conversión a cláusulas JPQL. Para lograrlo, este paquete se apoya en los servicios del sistema. A su vez, los servicios del sistema también invocan a este paquete, pues el punto de entrada de la búsqueda es simplemente un servicio más.

Adicional al paquete “backend”, se tienen 2 paquetes auxiliares:

Util: Clase que contiene métodos comunes que se pueden usar desde cualquier lugar de la aplicación. Un ejemplo de utilidad es el localizador de servicios de Spring, así como la generación de algoritmos de georeferenciación por medio del algoritmo de HAVERSINE.
Tool: Ejecutables que facilitan algunos procesos especiales del proyecto. Un ejemplo es un creador de lugares por consola, buscando facilitar el proceso desde base de datos directamente.

Modelo de dominio

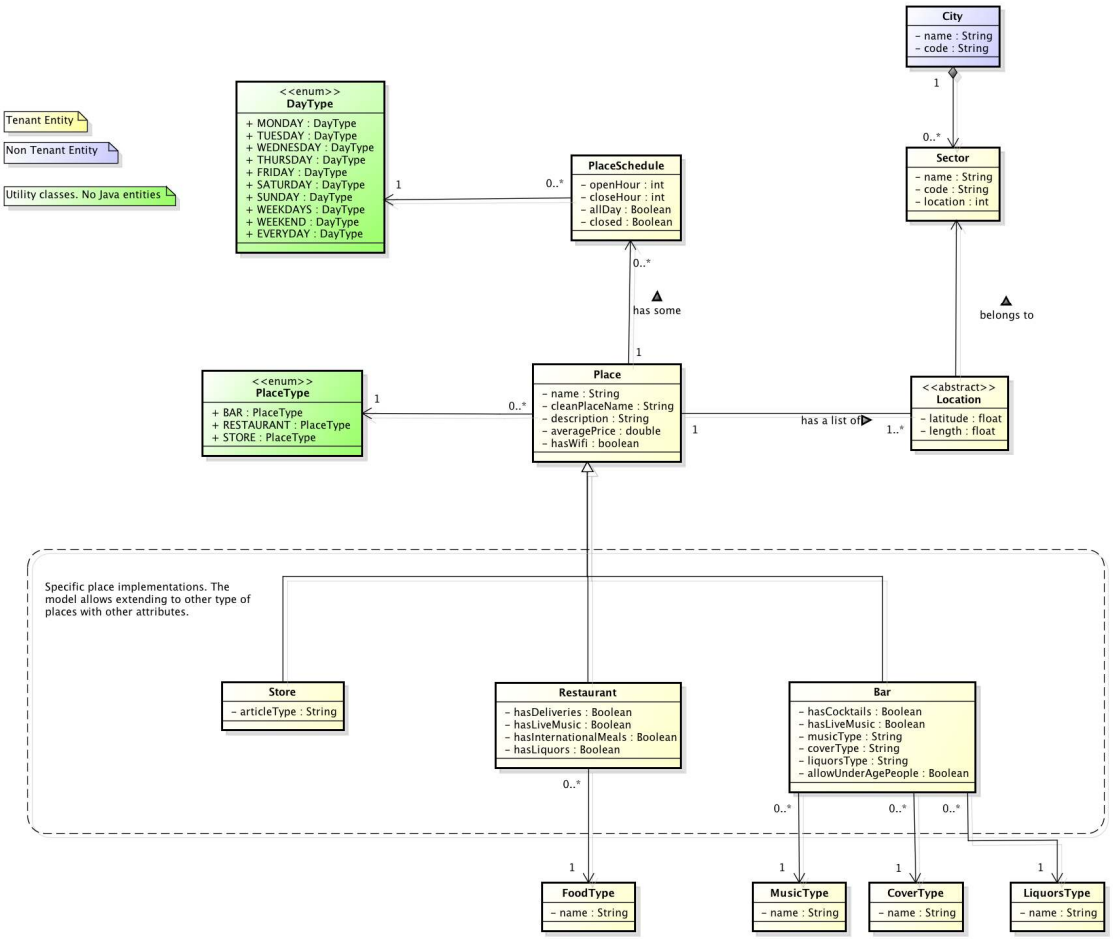

Ilustración 3: modelo de dominio

A continuación se exponen las entidades del sistema y se brinda una breve descripción de cada una. Sobre este modelo es que se aplican las búsquedas. Nuevamente vale la pena aclarar que el modelo es extensible y aunque para fines de presentación del proyecto se ha reducido a este modelo, no hay ninguna atadura a manejar esquemas más complejos.

Place: La clase place es la clase base del sistema. Un place representa a un establecimiento de cualquier tipo registrado en el buscador. En Place se manejan todos los atributos comunes a todos los tipos de establecimiento, como el nombre, precios, las ubicaciones, los horarios etc. En términos técnicos, un Place es una clase abstracta que no

puede ser instanciada, pero que permite definir una estructura general para todos los establecimientos de la aplicación. Un Place se puede identificar con un atributo “type”, que se obtiene del enumerador PlaceType.

Store, Restaurant y Bar: Implementaciones particulares de la clase Place. La aplicación soporta tantas implementaciones particulares como se desee. En cada clase se almacenan las particularidades en términos de atributos y de relaciones de cada tipo de lugar. Los atributos que tiene cada uno de los tipos de sitios son también parte del buscador. Es decir, se pueden filtrar restaurantes por el tipo de comida, por ejemplo. La extensibilidad del modelo se refleja en parte en la capacidad de agregar nuevos tipos de lugares sin modificar ninguna otra parte del modelo de dominio. Además, es posible crear tablas relacionadas para normalizar los atributos, como sucede con FoodType, LiquorType, CoverType, etc.

City: Una ciudad es un concepto muy importante en la aplicación, pues simboliza no solo una región, sino también una base de datos independiente del sistema. El color que diferencia la clase City de las demás significa que la entidad se almacena en una base de datos central, y no en una de las bases de datos por ciudad. Todo esto se discutirá en detalle en la sección de multitenancy.

Sector: Un sector significa un barrio, o una localidad de una ciudad. La relevancia de un sector dentro de la aplicación es que es una entidad sobre la que se generará un patrón. Toda ubicación particular de un establecimiento en el sistema, pertenece a un sector.

Location: Representa un punto en el mapa, donde se puede localizar un Place. Un Place, digamos, por ejemplo, McDonalds, puede estar ubicado en diferentes puntos de un mapa. Por lo tanto, un Place tendría una relación de uno a muchos con las ubicaciones

(Location). Un location trae inherentemente un sector, y unas coordenadas, lo que facilita la georeferenciación del Place.

PlaceSchedule: Para el manejo de horarios, un lugar podría abrir, digamos, los lunes de 8am a 2pm, los martes todo el día, y el resto de los días estar cerrado. Para permitir que el modelo soporte patrones de horarios, se ha creado esta entidad que encapsula una definición de horario. La entidad contiene un tipo de día (definido en el enumerador DayType), una hora en que se cierra y se abre el lugar (en formato de 24 horas), y dos operadores booleanos opcionales, uno para indicar si el sitio está abierto todo el día, y otro para indicar que éste se encuentra cerrado durante el DayType seleccionado (por ejemplo, cerrado en fines de semana, o abierto todos los martes)

3.6 Multitenancy

El concepto en general de multitenancy se refiere a “hospedar” múltiples clientes dentro del mismo software. Un cliente viene siendo un “tenant”, que usa el software y mantiene cierta independencia en sus datos y procesos con respecto a otros clientes.
Dentro de la aplicación, se adapta el concepto de multitenancy para referirse a diferentes ciudades. Buscando obtener un rendimiento óptimo ante grandes volúmenes de datos, cada ciudad pasa a ser un cliente , y se independizan ciertos aspectos entre ellas.
El modelo de multitenancy se basa en un manejo de múltiples bases de datos simultáneamente, donde cada ciudad tiene un propio lugar de almacenamiento de sus lugares, sectores, ubicaciones, etc. La idea con este modelo es disminuir el costo de acceso a datos, cuando se manejan diferentes ciudades. Aunque para este proyecto se eligió la ciudad como punto de separación entre bases de datos, es posible manejar otros criterios, como sectores, o países, etc.

Dado que los patrones están almacenados en base de datos, y algunas otras entidades no deben pertenecer a ninguna ciudad en particular, sino que son más bien globales a toda la aplicación, se cuenta también con una base de datos central.

El esquema de la base de datos central es diferente a la de cada ciudad, y todas estas son iguales entre sí. Al definir una entidad se debe registrar si esta entidad se debe utilizar en cada ciudad o en los “tenants” o en la central, es decir, en la base de datos “nonTenant”.

El estándar JPA define un concepto llamado PersistenceUnit. Una unidad de persistencia es un conjunto de mapeos objeto relacionales que se agrupan. En el momento de definir un EntityManager, o gestor de las operaciones de base de datos, se le asocia un PersistenceUnit determinado. Así mismo, una conexión a datos se asocia también a un PersistenceUnit, sin embargo, múltiples fuentes de datos pueden estar asociadas al mismo PersistenceUnit. Para el proyecto, se han creado dos unidades de persistencia: Una para los tenants y otra para la central.

JPA define la estructura de un archivo que debe estar en todas las aplicaciones, con nombre persistence.xml. Este archivo define las unidades de persistencia, y las entidades Java que pertenecen a cada una. En el siguiente gráfico se puede observar la estructura de una unidad de persistencia:

```
<persistence-unit name="nonTenantPersistenceUnit"
    transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- List of classes that do not depend of a particular tenant.
        This set of classes can't interact with others that depends on a tenant.
    -->
    <class>co.edu.eafit.meaningsearch.backend.entity.system.Parameter</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.city.City</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.AdjectivePattern</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.AttributeExpressionPattern</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.ComparatorPattern</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.NonMeaningWord</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.PlaceTypePattern</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.SchedulePattern</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.SchedulePatternValue</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.ScheduleDayReplacement</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.ScheduleHourReplacement</class>
    <class>co.edu.eafit.meaningsearch.backend.entity.search.pattern.ScheduleGenericExpression</class>

    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <shared-cache-mode>NONE</shared-cache-mode>
    <validation-mode>NONE</validation-mode>
    <properties>
        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
        <property name="hibernate.hbm2ddl.auto" value="create-update" />
    </properties>

</persistence-unit>
```

Ilustración 4: ejemplo de unidad de persistencia en persistence.xml

Como se puede observar, una unidad de persistencia simplemente agrupa un conjunto de entidades. En este caso, se puede observar la unidad de persistencia para clases que no pertenecen a un tenant. De la misma manera se ha definido una unidad de persistencia para aquellas entidades que pertenecen a todo tenant. Adicional a las clases, se definen algunas propiedades de cache, de dialecto de la base de datos, y de la herramienta HBM2DDL, que se explicará más adelante.

Una restricción que surge del manejo de múltiples unidades de persistencia es que es imposible relacionar entidades que pertenecen a diferentes unidades. Aunque esto puede parecer lógico desde el punto de vista de bases de datos, puesto que las bases de datos pueden estar en sitios totalmente separados e incluso, tener motores de base de datos diferentes, desde el punto de vista de Hibernate también es imposible utilizar la sintaxis JPA para relacionar entidades, lo que restringe la forma en que se realiza el CRUD de la aplicación.

Una vez se definen las entidades que pertenecen a cada unidad de persistencia, se deben definir las fuentes de datos, la transaccionalidad, etc. Para esto, se ha creado una estructura de archivos XML de contexto de Spring, que permiten extender nuevas ciudades con un costo en tiempo y complejidad muy bajo.

La estructura de los archivos de configuración de la aplicación se encuentra en el directorio src/test/resources, pues es donde, por defecto, maven localiza la configuración para pruebas unitarias. Éste es un vistazo a esta carpeta:

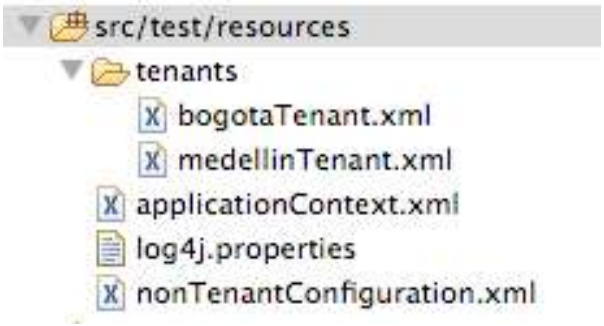

Ilustración 5: archivos de configuración

El archivo base de configuración es applicationContext.xml. Este define propiedades como la forma en que Spring identifica los componentes o servicios, y contiene referencias hacia otros archivos de contexto. Para efectos de multitenancy se ha definido un archivo llamado nonTenantConfiguration.xml, que contiene la configuración de base de datos para la central. Esta configuración incluye tanto la unidad de persistencia asociada, como los parámetros de conexión asociados a la base de datos.

De la misma manera, existe una carpeta llamada “tenants” donde se ubican todas las conexiones las bases de datos de cada ciudad. Para efectos del proyecto de grado se han definido 2 ciudades: Medellín (con id = 1) y Bogotá (con id = 2). Cada uno de estos archivos gestiona lo necesario para utilizar una base de datos de esta ciudad. Esta es la estructura del archivo de Medellín:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/

    <!-- JPA Entity Manager Factory Definition -->
    <bean id="entityManagerFactory1"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="tenantPersistenceUnit" />
        <property name="dataSource" ref="dataSource1" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="generateDdl" value="true" />
                <property name="showSql" value="true" />
                <property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect" />
            </bean>
        </property>
    </bean>

    <bean id="jpaTemplate1" class="org.springframework.orm.jpa.JpaTemplate">
        <property name="entityManagerFactory" ref="entityManagerFactory1" />
    </bean>

    <bean id="dataSource1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:8889/meaningsearchmedellin"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
</beans>
```

Ilustración 6: configuración para tenant Medellín

Sobre esta configuración es necesario mencionar ciertos detalles:

- En el bean llamado dataSource1 se definen parámetros de conexión para la base de datos de Medellín.
- En el bean entityManagerFactory1, se definen, entre otras cosas, el dialecto y la unidad de persistencia asociada a esta conexión
- Todos los beans tienen el sufijo “1”, puesto que en base de datos, Medellín es una ciudad con código 1.

La definición para Bogotá es análoga, simplemente con parámetros de conexión diferente y con sufijo “2”.

Existe un artefacto llamado MultitenancyFactory, que se encarga de cargar todos los beans de conexión de todas las ciudades. Esta clase escanea todas las ciudades existentes, y almacena en memoria (en estructuras de árboles) los parámetros de conexión.

En caso que se requiera agregar una nueva, ciudad, simplemente es crear un nuevo archivo de contexto y definir las propiedades particulares de esta conexión. Incluso podría usar un motor diferente de base de datos. Adicionalmente se debe insertar manualmente un registro que identifica a la ciudad en la base de datos central de la aplicación.

Hibernate contiene una herramienta llamada hbm2ddl, que en pocas palabras permite crear las tablas de una base de datos y sus claves primarias, foráneas, índices, etc. A partir de un modelo de dominio expresado en Java. Para la aplicación hacemos uso de esta herramienta, por lo que Hibernate genera todas las bases de datos (tanto la central como la de cada ciudad) con solo definir las conexiones de acceso a datos. Es también por esta razón por la que no se incluye en la presentación un modelo de base de datos, ya que es totalmente análogo al modelo de dominio Java. Los nombres de los atributos se convierten en nombres de columnas y las relaciones con listas de java se convierten en claves foráneas.

Para el tema de acceso a datos desde los servicios, se han creado 2 clases padres para los servicios, llamadas MultiTenantService y NonTenantService. Todo servicio debe extender de una de estas clases. Cada una de éstas define métodos como “save”, “createQuery”, etc. Con la particularidad que NonTenantService hace que estos métodos utilicen la base de datos central, mientras que MultitenantService hace uso, cada vez que se ejecuta un método, del código de ciudad asociado al hilo de ejecución, y a través de la clase MultitenancyFactory obtiene los objetos de conexión apropiados para ese hilo. De esta manera se consigue que los servicios no tengan que pensar sobre cómo obtener conexiones, ni que conexión usar. Simplemente utilizan los métodos de su clase padre, que encapsula toda esta lógica.

3.7 Introducción a los patrones

La definición de patrón en el proyecto de grado difiere de la definición general de un patrón en ingeniería de software. Me refiero, para el proyecto, a una expresión recurrente que se puede encontrar en un texto.

Desde un punto de vista técnico, un patrón es una clase Java. Generalmente, es de hecho una entidad de base de datos. Existen varios tipos de patrones que permiten cubrir diferentes tipos de sentencias.

El modelo de creación de consultas JPQL de la aplicación delega ciertas responsabilidades a los patrones. Un patrón de cierta forma es auto gestionado, pues es él mismo quien:

- Define si una cadena de texto cumple con su patrón, es decir, cumple las condiciones para que se aplique su cadena.
- Brinda algunas pautas que hacen pensar que es candidato para cumplir con un texto. Generalmente una palabra inicial es suficiente, para que se evalúe la expresión completa para ver si se cumple el criterio.
- Genera ciertas partes de la consulta JPQL. Un patrón, si se cumple en una cadena, proporciona una sentencia que hacer parte del Where de la consulta, o del Order By, o incluso puede indicar un Join con otra tabla. El "orquestador" simplemente usa cada patrón para ir generando un conjunto de "partes" de la consulta, y luego se apoya en una estrategia de generación de la consulta completa.

Como se puede observar, un patrón es una entidad crucial por muchos sentidos. Desde el diseño, un patrón comienza simplemente con una interfaz de un simple método:

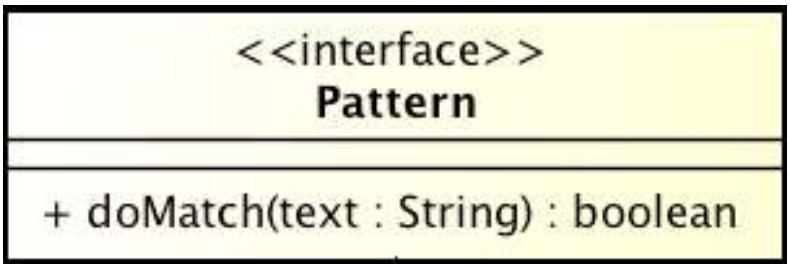

Ilustración 7: Declaración de la interfaz Pattern

Todo patrón debe tener la forma de validar si un texto cumple o no con él. Algunos patrones deben recibir toda la expresión, o en otros simplemente una palabra es suficiente.

Como se mencionó, cada patrón puede generar diferentes partes de una sentencia JPQL. Antes de entrar a detallar cada tipo de patrón, y su funcionamiento, conviene explicar el modelo de cláusulas JPQL y la misión de los creadores de cláusulas.

3.8 Modelo de sentencias JPQL

Aunque explicar el modelo del lenguaje JPQL en detalle está por fuera del alcance de este proyecto, a continuación se explicarán algunas generalidades de esta sintaxis:

- Es un lenguaje que busca conservar las ventajas de orientación a objetos cuando se habla de bases de datos.
- Las consultas aplican sobre clases y atributos, y no sobre tablas y columnas.
- Se abstraen las particularidades comunes de un motor de base de datos, lo que brinda una transparencia para el desarrollador.

Algunos ejemplos de consultas JPQL en el modelo de dominio son:
“Select p from Place p”: El resultado es una lista de todos los objetos Place que se encuentran en el sistema.
“Select p, l from Place p LEFT OUTER JOIN p.locationList as l”: El resultado es una lista de todos los objetos Place y sus respectivas ubicaciones. El retorno de esta consulta es una lista de arreglos, donde cada ubicación de la lista es un arreglo de 2 posiciones, el objeto Place en la primera posición y una lista de Location como segunda posición.

El estándar JPA utiliza ciertas anotaciones a nivel de clase que se deben incluir en las entidades. Estas anotaciones no solo permiten realizar una correspondencia entre clases y tablas, sino que también permite realizar relaciones entre entidades (que a la larga, por medio de la herramienta hbm2ddl se convierte en tablas foráneas del modelo de base de datos.
Esta, por ejemplo, es la una parte de la definición de la entidad Location:

```java
@Entity
@Table(name = "LOCATION")
public class Location extends BaseEntity {

    /**
     * Serialization ID
     */
    private static final long serialVersionUID = 5764268679142354430L;

    /**
     * Unique identifier of the register
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Column(scale = 10, precision = 6)
    private Float latitude;

    @Column(scale = 10, precision = 6)
    private Float longitude;

    @ManyToOne
    @JoinColumn(name = "sectorId", nullable = false)
    private Sector sector;

    @ManyToOne
    @JoinColumn(name = "placeId", nullable = false)
    private Place place;
```

Ilustración 8: Uso de anotaciones JPA en la entidad Location

De esta porción de código quisiera hacer énfasis ciertos detalles:

- Todo comienza con la anotación "@Entity". Sin esta anotación no se identifica que esta es una clase que será gestionada para consultas con JPA.
- La anotación @Table permite definir de manera explícita el nombre de la tabla que representa esta clase.
- El entero id se define como clave primaria de la tabla con la anotación @Id, además, se define una generación automática de los elementos de esta columna con la anotación @GeneratedValue.

- Las relaciones con otras entidades también se definen por medio de anotaciones. Por ejemplo, Location contiene una relación con el Place al que pertenece. Esta relación es Muchos a Uno (ManyToOne). La anotación que se define permite no solo explicar el tipo de relación con la entidad Place, sino que además define una columna (en este caso placeId) que será donde se ubicará el id del Place en la tabla de Location. Cuando la herramienta hbm2ddl inspecciona las anotaciones, genera tanto la tabla, la columna, la clave primaria (con los campos marcados con @Id), así como las claves foráneas desde el campo placeId hasta la tabla definida para la entidad Place.
- Todo atributo de la entidad, por defecto, se considera una columna persistible. Es posible indicar que un atributo no se desea persistir por medio de la anotación @Transient. En este orden de ideas no es necesario marcar los atributos con la anotación @Column. Sin embargo, es posible indicarle a Hibernate ciertas propiedades que deben cumplir las columnas de base de datos generadas. En este caso, las coordenadas de latitud y longitud de la entidad Location requieren ciertos atributos de número de decimales almacenados, por lo que en la anotación @Column se define tanto la escala como la precisión, que tienen el mismo significado que en una base de datos regular.

Es necesario recordar que Hibernate contiene el concepto de dialecto, que permite abstraer las particularidades de algún motor de base de datos, y maneja de manera transparente la definición del modelo de datos. Es decir, que la “traducción” que se hace de estas anotaciones puede diferir entre motores de base de datos. Sin embargo, desde el punto de vista del desarrollador Java, esto es transparente.

3.9 Modelo de cláusulas

El siguiente es el diagrama objetual que representa las cláusulas y sus atributos:

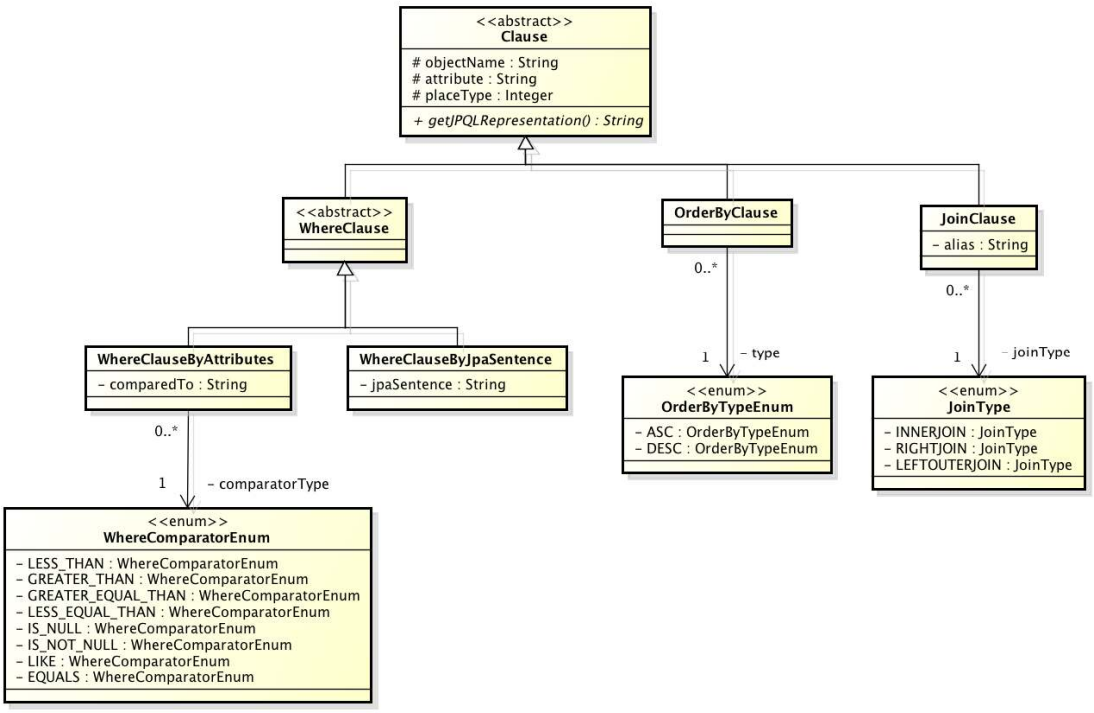

Ilustración 9: Esquema de cláusulas de la aplicación

Como se puede observar, todo parte de la clase abstracta Clause, y 3 subclases: WhereClause, OrderByClause y JoinClause. A continuación se da una breve descripción a cada entidad:

Clause: Clase base de toda cláusula en el sistema. Esta clase contiene ciertos atributos comunes a todas las cláusulas, como el nombre del objeto y atributo sobre el que opera. Por ejemplo para la sentencia JPQL "SELECT p FROM Place p WHERE p.averagePrice < 50000" se generaría una cláusula WHERE donde el nombre del objeto es p y el atributo sobre el que opera es averagePrice.

Además, existe un atributo placeType que es muy importante para mantener la consistencia de las consultas. Existen ciertas comparaciones o atributos que no aplican sobre Place sino sobre una de sus propiedades específicas. Es imposible por ejemplo, ordenar diferentes tipos de sitios por una propiedad que solo aplica a uno de ellos. Ese

tipo de “restricciones” se definen en este atributo. El generador de consultas interpreta las restricciones entre las cláusulas generadas y garantiza la consistencia entre ellas.
La clase Clause también define un método abstracto llamado getJPQLRepresentation, que en cada una de sus subclases debe implementarse. La misión es generar un String que representa la información de la cláusula en lenguaje JPQL. No es una consulta completa, sino simplemente la porción que representa en la consulta final.

WhereClause: Cláusula que representa una porción del Where de la consulta a generar. Existen 2 formas de realizar una cláusula Where. La primera es por medio de la clase WhereClauseByAttributes, que simplemente define las condiciones de la cláusula por medio de atributos Java, y las une con un tipo de comparación (igualdad, mayor que, etc.). La segunda es definiendo como String la cláusula JPA. Esto es útil para algunos patrones que insertan porciones de la cláusula Where en base de datos, y no es fácil definir todos sus atributos.

OrderByClause: Simboliza una parte de una cláusula OrderBy. Esta cláusula simplemente menciona un atributo por el que se debe ordenar, y un tipo de orden, bien sea ascendente o descendente.

JoinClause: Algunos patrones requieren definir porciones de una cláusula Join, buscando incluir en una cláusula Where entidades diferentes a Place (por ejemplo, para patrones que operan sobre la entidad PlaceSchedule). Buscando cumplir con esta situación, se provee un soporte para cláusulas Join.

3.10 Creadores de cláusulas

Un patrón generalmente crea una cláusula. Sin embargo, algunos tipos avanzados de patrones crean más de una cláusula. Buscando proveer una flexibilidad en la generación

de cláusulas por parte de los patrones, se crea la figura de ClauseCreator. Esta es una interfaz – con interfaces hijas – que define la capacidad de un patrón de generar cierto tipo de cláusulas. En términos UML a continuación se presenta la definición del creador de cláusulas:

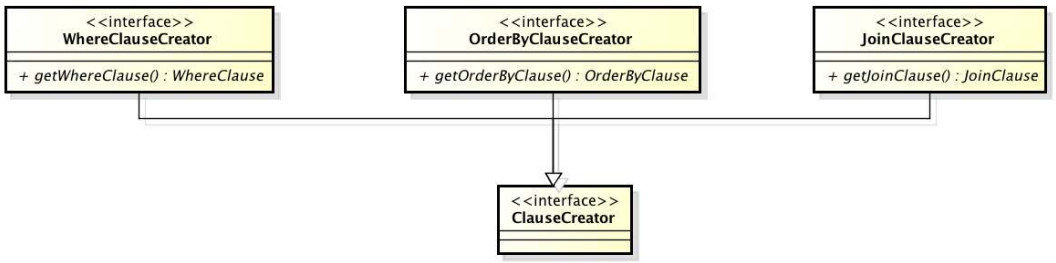

Ilustración 10: creadores de cláusulas

Como se puede observar, existe una interfaz de creación de cada tipo de cláusula. Cada patrón decide que interfaces de creación de cláusula implementar, y con ello, permite aportar a la consulta resultado en diversas maneras, dependiendo de la naturaleza del patrón.

3.11 Tipos de patrones

Los patrones a un nivel técnico se organizan en 3 grupos. Cada uno de estos grupos es una clase Java de la que deben heredar.

Patrón de palabra simple: En código llamado SimpleWordPattern, representa un patrón que busca coincidencias de una palabra específica. Si la palabra se encuentra en el texto a buscar, se toman determinadas acciones especificadas por el tipo concreto de patrón.

Patrón basado en servicios: Existen ciertos patrones en los que las opciones de coincidencia son definidas por alguna tabla de base de datos. Para el caso del proyecto de

grado se manejan los sectores (o barrios) como ejemplo de este comportamiento. Cuando en la búsqueda se ingresa algo tipo “sitios cerca de Manrique” se debe reconocer Manrique como un barrio si este se encuentra en base de datos, y se deben filtrar los resultados solo para lugares que tengan una ubicación en el barrio Manrique.

Patrón basado en expresiones regulares: Quizás la forma más avanzada de un patrón es aquella donde las condiciones de “Matching” y el resultado final no se pueden determinar sin tener la consulta presente. En aquellos tipos de expresiones donde algunos valores de la entrada (la consulta) son a su vez valores a tener en cuenta (de manera transformada) en la salida, son patrones basado en expresiones regulares. Aunque se explicará en detalle los patrones de este tipo, un ejemplo para tener en cuenta mientras tanto es por ejemplo “sitios abiertos los martes de 8am a 2pm”. No es práctico tener este patrón como uno de palabra simple (se debería crear un patrón diferente para cada hora) por lo que una expresión regular que capture todos los posibles días y posibles horas, es lo idea. Además, como se puede observar, esta consulta se transforma en una cláusula Where que debe contener valores dinámicos (martes, 8am, 2pm).

Todo patrón de palabra simple se podría modelar como patrón por expresiones regulares. Sin embargo estos últimos tienen un rendimiento mucho menor pues utiliza el API de expresiones regulares de Java. Mientras sea posible modelar un patrón como de palabra simple, es recomendable hacerlo.

3.12 Patrones de palabra simple

Patrones de tipo de lugar: En código representado en la clase PlaceTypePattern, es un patrón que de acuerdo a una palabra, filtra el tipo de lugar para los resultados a obtener. La lista de patrones se obtiene de base de datos, por lo que esta clase es a su vez una entidad persistible. Contiene los siguientes campos:

Expresión: String que simboliza la palabra, o frase, que identifica a un tipo de lugar.

Type: Entero que identifica el tipo de lugar por el que se debe filtrar, en caso que la expresión se encuentre en el texto.

Se pueden ingresar tantos patrones de tipo de lugar como se desee. Cuando la aplicación comienza, todos estos patrones se cargan en estructuras de árboles para tener una búsqueda más eficiente.
Este patrón implementa la interfaz WhereClauseCreator, por lo que en caso que la expresión se encuentre en el texto a buscar, se genera una cláusula Where donde indica el tipo de sitio que debe tener la búsqueda.

Patrones de adjetivos: Un patrón de adjetivos (O AdjectivePattern en el código) es una expresión que simboliza una cualidad de un sitio. Con cualidad nos referimos a una forma de calificar y comparar sitios. No es, en ningún momento, un filtro. Simplemente dan una idea de los resultados que interesan obtener primero en una consulta.
Un patrón de adjetivos es también un patrón de palabra simple. Ejemplos de expresiones de patrones de adjetivos son “barato”, “caro”, “bonito”, “exclusivo”, “nuevo”, “antiguo”. Como se puede ver estas expresiones son relativas, es decir, no excluye algún sitio, solo indican que tipo de sitios se desea observar primero.
Dentro del modelo de cláusulas, este patrón implementa la interfaz OrderByClauseCreator, donde indica una expresión por la que se desea ordenar los resultados.
Los campos que contiene un patrón de adjetivos son los siguientes:

Expresión: Expresión que hace que se cumpla el patrón. Por ejemplo, “barato”.

Tipo de lugar: Algunas expresiones hacen que solo se pueda aplicar sobre algún tipo específico de lugar. Esto es común cuando se quiere realizar un Order By sobre una propiedad que pertenece a una subclase de Place, y no a la clase padre. En ese caso se indica en este campo el tipo de sitio sobre el que se debe crear la consulta. Es responsabilidad de otra clase interpretar estas restricciones de tipo de sitio.

Propiedad: Propiedad por la que se debe ordenar en caso que el patrón cumpla.

Orden: Indica el tipo de orden que se debe realizar sobre la propiedad, ya sea ascendente o descendente.

Patrones de atributos: Representado como AttributePattern en Java, es un patrón que simboliza el valor que debe tener una propiedad particular de un sitio. Éste es otro tipo de expresión de palabra simple. Ejemplos de estos patrones son por ejemplo, “cocktails”, que indica que la propiedad hasCocktails de la subclase Bar debe ser “true”.
Los campos de este patrón son:

Expresión: Como en los demás, indica la palabra que hace que aplique el patrón.

Tipo de lugar: Indica si existe una restricción sobre el tipo de lugar a buscar. Esto es muy común cuando, como en el ejemplo dado de cocktails, la propiedad existe para solo una de sus subclases.

Propiedad: Propiedad de la clase (o subclase) por la que se va a filtrar.

Valor: Valor que debe tener esa propiedad para que un resultado se considere válido para esta cláusula.

Este patrón implementa, de igual forma, la interfaz WhereClauseCreator, por lo que se traduce en una parte del Where de la consulta a generar.

3.13 Patrones basados en servicios

Tipo de patrón en el que la "expresión" que hace que se cumpla o no un patrón, se encuentra en una tabla independiente de base de datos.

En muchos casos es imposible determinar patrones puesto que estos se basan en sustantivos (como un patrón por nombres, apellidos, ciudades, etc.) que no son palabras fijas. Si estas tablas se encuentran en una tabla del modelo de datos, es posible crear patrones que guardan en un cache los registros de la tabla y buscar sobre ese caché. Generalmente los patrones basados en servicios, cuando encuentran que una porción del texto de búsqueda coincide con un patrón, crean una cláusula where donde se filtra por el patrón encontrado.

Existe una clase base llamada ServiceBasedPattern. Esta clase define un esquema para todas las implementaciones particulares de patrones basados en servicios. Pero fuera de definir los métodos, también define un comportamiento para refrescar el caché que maneja cada patrón. La clase ServiceBasedPattern define una tarea programada para refrescar el caché cada determinado tiempo (El tiempo para refrescar se almacena como un parámetro de la aplicación). Cada patrón, no obstante, define las consultas para refrescar el caché, y define la cláusula particular cuando se encuentra una palabra definida en el caché.

Para el proyecto de grado se define un patrón basado en servicios de ejemplo: SectorPattern.

Patrón de sectores: Este patrón se apoya en el servicio SectorServices para obtener todos los sectores de la aplicación. Los nombres de los sectores se almacenan en estructuras de árboles. En caso que un texto a buscar contenga el nombre de un sector, se

crea una cláusula Where donde se filtra que al menos una de las ubicaciones de los lugares a retornar, se encuentren en el sector encontrado.

Por ejemplo, si un texto contiene la expresión “parque lleras”, el sistema encuentra que este es un sector, si se encuentra en base de datos, y filtra los resultados a solo los que se encuentren en el parque lleras (crea una cláusula Where que lo especifica).

Las búsquedas sobre patrones basados en servicios nunca se realizan contra la base de datos real por temas de rendimiento. Solo se realizan sobre los datos en caché. De ahí la importancia de manejar tiempos cortos para refrescar el caché.

3.14 Patrones de expresiones regulares

Los 2 tipos de patrones mencionados proveen la capacidad de crear consultas complejas basados en palabras fijas, o que se encuentran en base de datos. Sin embargo, existen diferentes tipos de expresiones humanas en las que algunas palabras son fijas, pero otras contienen valores que varían y determinan la consulta a hacer.

Ejemplos de este tipo de expresiones son: “comida de menos de 30000”, donde se encuentran ciertas expresiones fijas (“comida”,“menos”) y a la vez una expresión que debe formar parte de la cláusula resultante (30000).

Este tipo de expresiones plantean varios retos. En primer lugar, se deben manejar variables que se conviertan a partes de la cláusula resultante. Además, por ejemplo, tal vez el usuario no escribe 30000, sino “treinta mil”, lo que hace que una consulta sobre enteros sea imposible de hacer. Esto resulta en que se deba realiza cierto “reconocimiento y transformación de variables” antes de insertarlas como parte de una consulta.

Los patrones de expresiones regulares permiten tener flexibilidad sobre las consultas, abren miles de oportunidades de patrones, y lo más importante, utiliza la misma consulta para generar valores de salida. En contra tiene una cierta complejidad para ingresar nuevos patrones.

Para el proyecto de grado se han implementado 2 tipos de patrones de expresiones regulares: Patrón de horarios y patrón de comparaciones numéricas.

Patrones de horarios: Este es el diagrama general de patrones de horario:

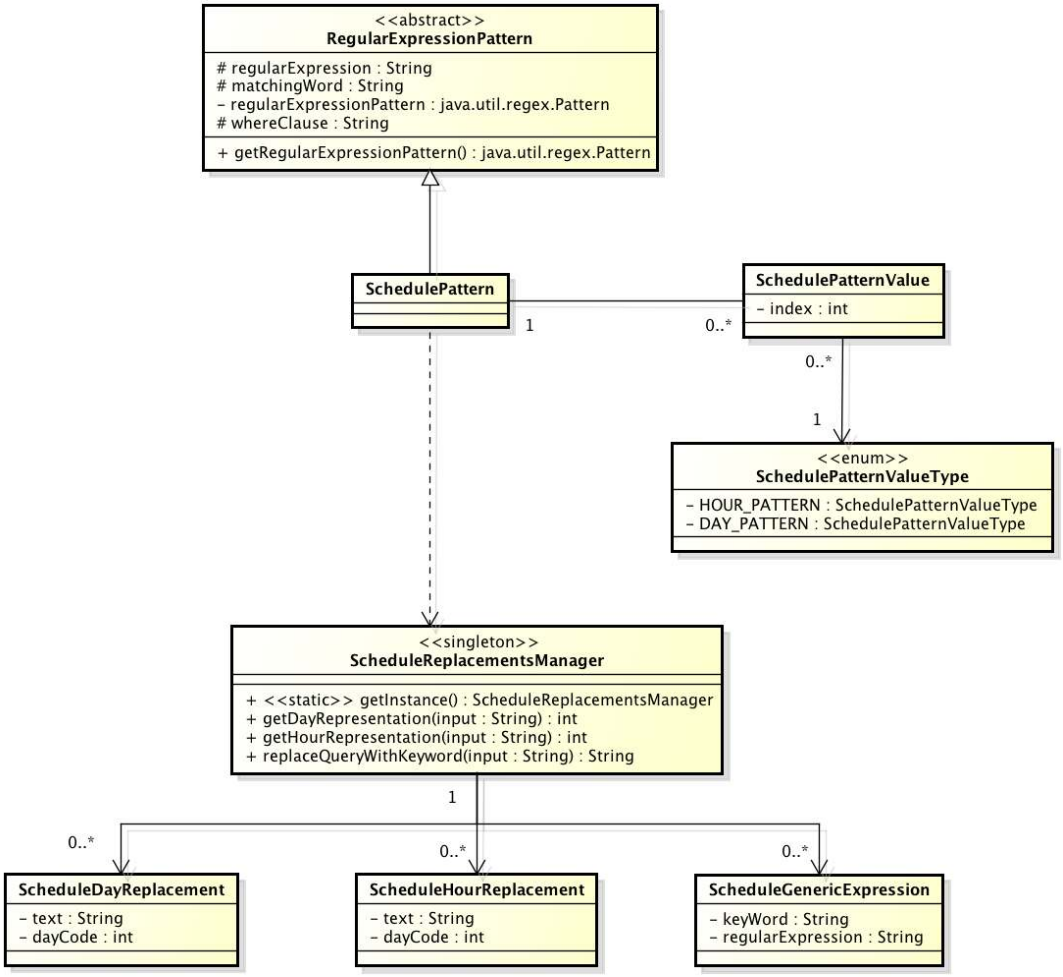

Ilustración 11: Estructura general de patrones de horario

El modelo de dominio contempla el manejo de horarios para los lugares. Existe una entidad llamada PlaceSchedule definida con los siguientes atributos:

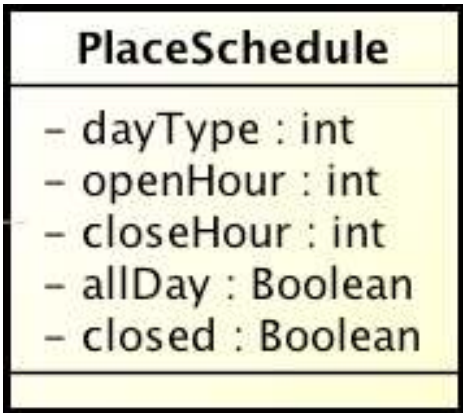

Ilustración 12: Definición de la entidad PlaceSchedule

DayType es un tipo de día definido por un enumerador con el mismo nombre. No solo existe un número equivalente a “lunes”, “martes”, etc. Sino que también se contemplan tipos de día como “fines de semana”, “todos los días”, etc.

Para cada registro PlaceSchedule se puede manejar un rango de hora en que el lugar abre y cierra, o se puede marcar, por medio de unas variables booleanas, si el sitio está abierto todo el día, o si ese día no abre.

Los patrones de horarios facilitan la detección de objetos PlaceSchedule que cumplan con los criterios de la búsqueda. Estos patrones crean tanto una cláusula Join como una Where para poder utilizar objetos PlaceSchedule.

Existe una entidad llamada SchedulePattern que representa un patrón de horario. Existe un atributo del patrón que representa a una expresión regular en lenguaje un poco más entendible.

Y esto pasa por que la idea es que el usuario final no deba conocer la sintaxis específica de las expresiones regulares en Java. Por tanto, al momento de cargar las expresiones se realiza un reemplazo de ciertas palabras claves en expresiones regulares.

Por ejemplo, el siguiente es un ejemplo de una expresión regular en la aplicación:

Abierto desde las #ANY# hasta las #ANY#

Esta expresión es mucho más legible por una persona que quiera insertar un patrón. Cuando se cargan las expresiones, se reemplaza la expresión “#ANY#” por:

(([a-z]|[A-Z]|[0-9]|)+)

Y de esta manera se permite utilizar expresiones regulares comunes de Java. Cada uno de estos valores “variables” pueden o no ser usados como parte de una cláusula Where. Sin embargo, las expresiones, como se habían mencionado, pueden estar en un formato que es necesario traducir antes de ingresarlo a una cláusula. Por esto se introduce el papel de ScheduleHourReplacement y ScheduleDayReplacement.

Para el caso de patrones de horario, un #ANY# podría representar o una hora del día o un día de la semana. Para cada uno se contabiliza una serie de reemplazos (por ejemplo: “8 de la noche” se convierte en 20) que permitan convertir palabras coloquiales, o textos, en campos que sean comparables y que se entiendan en el contexto de la base de datos. Estas 2 entidades, entonces, contiene reemplazos a hacer tanto para días como para horas. Como se encuentran en base de datos, es posible en cualquier momento ingresar nuevos registros para reconocer nuevos patrones (Se debe implementar una forma de refrescar el caché, puesto que ninguno de estos reemplazos suceden contra la base de datos real).

Para identificar si un #ANY# representa una hora o un día, se usa la entidad SchedulePatternValue, que almacena para cada uno, que tipo de reemplazo efectuar. Estos tipos están definidos en la entidad SchedulePatternValueType.

Adicionalmente, se brinda la posibilidad de convertir palabras claves en expresiones regulares, por medio de la entidad ScheduleGenericExpression.

Ahora que se ha explicado el modelo, se debe explicar tanto la forma en que se realiza la operación de “doMatch”, y la generación de la cláusula Where.

Por temas de rendimiento, toda cláusula debe tener una palabra clave llamada matchingWord. Solo se evalúa una expresión regular, si el texto a contener contiene esta palabra, que lo hace "candidato" a ser un patrón. El método doMatch evalúa toda la consulta para ver si cumple con los criterios de expresión regular. En caso que se cumpla, sigue siendo tan solo un candidato, pues es necesario interpretar cada uno de los valores de runtime (los #ANY#) y validar que puedan ser reconocidos y usados en una cláusula WHERE. En caso que todos los valores dinámicos se reconozcan, la expresión aplica.
Para la generación de la cláusula WHERE, se utilizan sentencias JPA con valores dinámicos. Para el ejemplo dado de "abierto desde las #ANY# hasta las #ANY#", esta sería la sentencia WHERE que se define en el patrón en base de datos:

#TODAY# AND schedule.allDay = true OR (schedule.openHour < #0# AND schedule.closeHour > #1#)

#TODAY# es una palabra clave que se reemplaza por una parte de la consulta en la que se indica que el DayType aplique (sea hoy, o todos los días, o fines de semana o días de la semana, dependiendo del caso).
Los valores #0# y #1# indican reemplazos en tiempo de ejecución de los #ANY# del patrón. El 0 o el 1 representan la posición del ANY a reemplazar. El valor que se ubica en la cláusula WHERE no es el de la consulta, sino el que fue convertido por medio de las entidades de reemplazo, por lo que la consulta es entendible para JPQL.

Patrones de comparaciones numéricas: Este patrón usa expresiones regulares para manejar comparaciones numéricas, generalmente de precios, sobre los lugares. Las expresiones regulares son muy parecidas a las de los patrones de horario, pero en esta ocasión se reconocen solo números, por medio de la palabra clave #NUMBER#. Un ejemplo de una expresión regular de comparaciones numéricas es:

de menos de #NUMBER#

La expresión #NUMBER# se reemplaza por una expresión regular que verifica cualquier número, ya sea entero o con decimales. Los patrones de comparaciones numéricas simplemente producen una cláusula WHERE donde se encuentran, de la misma manera que en los patrones de horario, expresiones que contienen palabras claves que se reemplazan en runtime.
Por ejemplo, para la expresión regular planteada, la siguiente es la expresión WHERE que se almacena en base de datos y que será reemplazada por el valor que ingrese el usuario:

p.averagePrice < #0#

El espacio ocupado por #0# es reemplazado por el número que el usuario ingrese. Esto provee la flexibilidad para crear patrones para comparaciones de precios, y cualquier otra entidad numérica que se soporte en el dominio del sistema.

3.15 Soporte para geolocalización

La idea de esta funcionalidad es retornar lugares cercanos a cierto punto del mapa. Para lograrlo, se usó el algoritmo de Haversine, que permite encontrar distancias entre 2 puntos de los que se conoce su latitud y longitud.
A la aplicación es posible indicarle que se filtren los resultados con un determinado rango, indicándole una latitud y longitud. Sobre estas coordenadas se genera una cláusula WHERE que contiene la expresión de Haversine, y que por medio de cosenos y tangentes garantiza que los sitios a retornar solo tendrán un determinado rango de distancia con una coordenada dada.

3.16 Flujo de búsqueda

El orquestador de todos los elementos mencionados es la estrategia de búsqueda. Una estrategia es simplemente un servicio que convierte un texto cualquiera en una consulta JPQL completa, haciendo uso de búsqueda de patrones.

La estrategia implementada para el proyecto de grado mantiene todos los patrones en estructuras de árboles por medio de mapas. Estos mapas contienen como clave la palabra "indicio" de que un patrón aplica.

La estrategia analiza cada uno de los patrones candidatos, y aquellos que aplican para el texto ingresado se van agregando a una lista. Posteriormente, de cada uno de esos patrones se obtienen las cláusulas apropiadas, y con ellas se genera una expresión JPQL completa.

Para hacerlo, Se utilizan varias herramientas auxiliares:

PatternToClausesConversorUtility: Clase que abstrae todas las cláusulas posibles de una lista de patrones, y las almacena en estructuras convenientes para generar el JPQL completo.

ClausesGrouperUtility: Las cláusulas que afectan la misma propiedad se agrupan dentro de una sola cláusula WHERE con operadores OR. Esta clase permite generar expresiones agrupadas.

JpaSentencesGeneratorUtility: Herramienta que convierte una lista de patrones en una cláusula completa JPQL. Recibe las cláusulas de manera individual con ellas crea un String completo que se ejecuta sobre la base de datos.

El flujo completo de la estrategia se puede observar en el siguiente diagrama de secuencias:

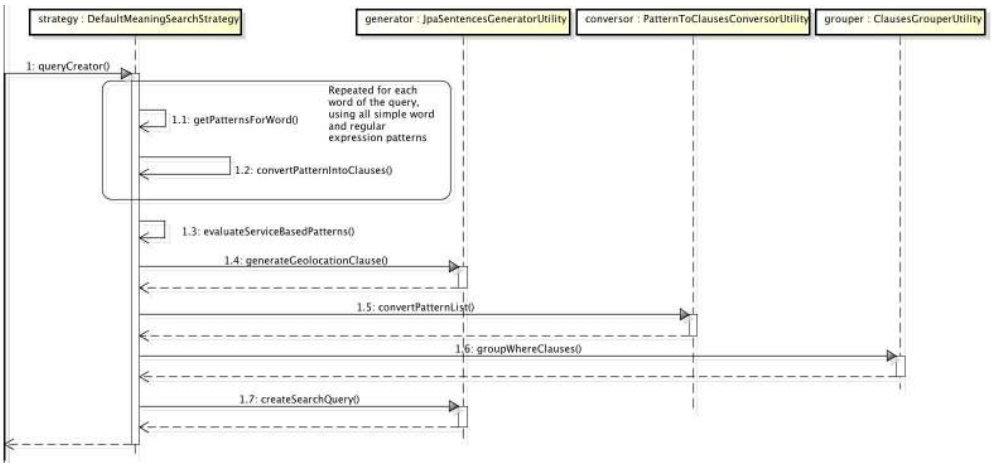

Ilustración 13: Estructura general de búsqueda y creación de cláusulas

Como se puede observar, el proceso es sencillo, consiste en analizar los patrones candidatos sea para cada palabra del modelo, o para toda la consulta. Los patrones se convierten en cláusulas, y se agrega, si aplica, la cláusula de geolocalización. A continuación se agrupan las cláusulas, y luego se crea la consulta. De esta manera se pueden construir consultas complejas, que agrupan varios operadores lógicos, y se convierte en una lista de Place del sistema.

3.17 Instalación

El proyecto de grado incluye un conjunto de pruebas unitarias que se pueden ejecutar desde cualquier IDE Java. Todo el proyecto fue desarrollado bajo la herramienta eclipse. Los archivos de configuración nonTenantConfiguration.xml, medellinTenant.xml y bogotaTenant.xml, son los que contienen los parámetros de conexión a las diferentes bases de datos. Es necesario que estos parámetros sean configurados adecuadamente.

Las bases de datos se deben crear manualmente. Sin embargo, una vez se crean, la herramienta genera automáticamente todas las tablas. Para esto, es necesario ejecutar el

set de pruebas una vez. Las pruebas van a fallar, puesto que las bases de datos se encuentran vacías.

Para insertar los patrones, se entregan unos archivos de script que deben ser insertados en la base de datos central, y otro que tiene un conjunto de lugares para realizar las pruebas. Se deben ejecutar los comandos en la base de datos, y el sistema con esto puede ejecutar las pruebas.

Para hacerlo en eclipse, se debe dar click derecho en la clase de pruebas que se desea ejecutar, “run as”, y seleccionar “Junit test”.

3.18 Manejo de librerías

La gestión de librerías se lleva a cabo usando la herramienta Maven. Esta permite gestionar versiones de las librerías, los administra en un repositorio local, y configura la gestión de las mismas en el entorno Java.

Todas las librerías usadas son libres y no requieren ningún tipo de licenciamiento. Para observar un resumen completo de todas las librerías usadas y sus versiones, por favor abrir el archivo pom.xml en la raíz del código fuente.

3.19 Pruebas unitarias

Para efectos de demostración de los resultados del proyecto, se han creado un conjunto de pruebas unitarias basadas en Junit. La metodología seguida para las pruebas es la siguiente:

- Se crea un conjunto de lugares, de forma predefinida, en forma de scripts. Estos lugares son aquellos sobre los que se debe buscar.

- Se insertan en base de datos un conjunto amplio de patrones, de todos los tipos, que permiten realizar búsquedas complejas sobre los lugares a buscar.
- Se crean diferentes clases, de pruebas unitarias, de acuerdo a la naturaleza de las pruebas que se quieran tener (sobre patrones específicos, combinando, etc.)
- Se crean métodos de pruebas unitarias. Cada método debe satisfacer lo siguiente:
    - Verificar que la consulta JPQL retornada sea la adecuada para la búsqueda
    - Obtener los resultados de lugares, e imprimirlos para efectos de presentación.

Las pruebas realizadas son las siguientes, por cada una se presupone que se realizan las verificaciones pertinentes en código:

- Verificar que los patrones de tipo de lugar funcionen correctamente. Esto debe realizarse por medio de una consulta con la palabra “bares”, donde bares quiere decir que solo entregue lugares de tipo “bar”.
- Validar los patrones de adjetivos, por medio de 2 tests, “sitios baratos” y “sitios caros”. La lista retornada debe ser la misma, pero en orden inverso. Verificar este orden inverso.
- Validar que los patrones de atributos funcionen, por medio de 2 consultas: Una sobre “tiendas de camisetas”, debe retornar tiendas cuyo tipo de artículo sea “camisetas”. Aquí se demuestra además que se pueden realizar consultas sobre atributos de solo un tipo de sitio. Además, se consultan “sitios con wifi”, que es una propiedad de Place.
- Validar que los patrones orientados a servicio funcionen correctamente. Ejecutar una consulta donde se especifique “sitios alrededor del parque lleras”, donde “parque lleras” coincide con un sector en base de datos. Verificar los resultados de la consulta.

- Ejecutar una consulta donde se demuestre la funcionalidad de patrones de horario. Para esta primera prueba se debe buscar “sitios abiertos los martes de 10am a 9 de la noche”. La consulta debe contener una expresión JOIN con la entidad PlaceSchedule, y los sitios retornados deben contener esta particularidad.
- Demostrar que los patrones de comparación numérica funcionan, por medio de la ejecución de la búsqueda “sitios donde el precio sea menor a 40000”. Los resultados se deben filtrar por esta restricción.
- Para los patrones de expresiones regulares, también se debe probar que si, el texto “variable” no es reconocido, la expresión no se debe tomar en cuenta. Por ejemplo, la búsqueda “sitios abiertos desde las 4pm hasta las Medellín” no debe retornar nada, pues carece de sentido lógico la expresión “Medellín” en ese momento. Las variables se validan contra unas expresiones regulares y patrones de reemplazo, y esta no debe cumplir.
- De igual manera realizar un test negativo sobre patrones de comparación numérica, en donde la expresión a buscar no sea un número, y probar que no se utiliza en la consulta.
- Por último se realizarán 4 pruebas donde se combinan los patrones, para demostrar la habilidad de crear consultas complejas. Los patrones de palabra simple y de expresiones regulares también pueden ser combinados. Además, probar que cuando las búsquedas involucran atributos particulares de 2 tipos de sitios diferente, estos 2 son descartados y solo prevalecen aquellos que son generales a todos.

Los resultados de estas pruebas pueden verse en “vivo” durante la presentación. Se adjuntan los resultados de las pruebas en el formato de respuesta de jUnit para Eclipse:

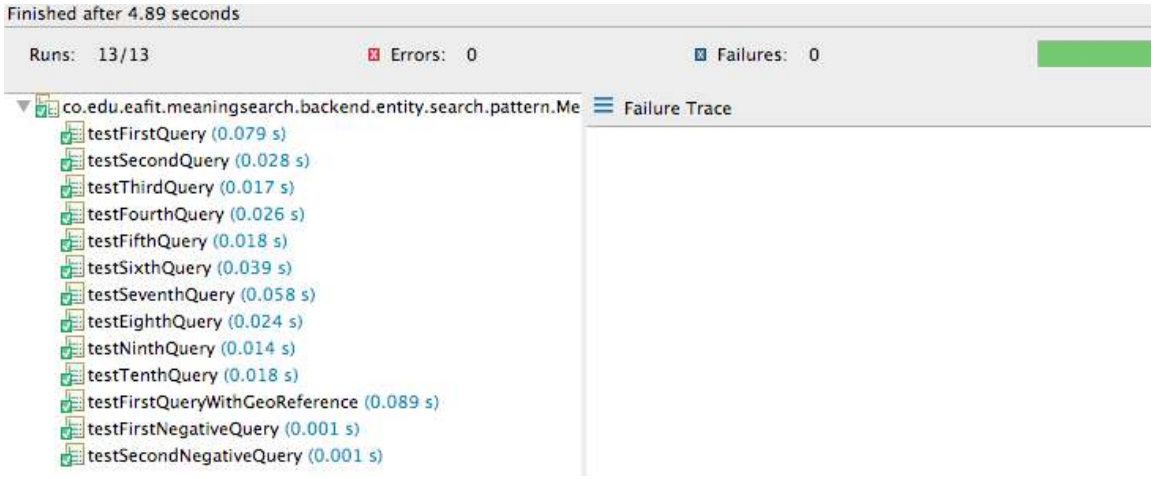

Ilustración 14 Resultados de ejecución de pruebas unitarias

4 TRABAJOS FUTUROS

- Una limitante que puede presentar el sistema es cuando se utilizan nombres propios (nombres de lugares, sectores, etc.) que coinciden con algún patrón, pero que en realidad no se refieren a el. Si existe un sector que contiene un nombre que a su vez hace Match en otro patrón, la búsqueda no va a devolver lo que deseaba el usuario. Es deseable entonces, permitir que el usuario descarte, o defina, que partes de la oración se deben tomar como candidatas para patrones y cuales no, o especificar las cláusulas encontradas , de manera que se pueda refinar la búsqueda agregando o descartando patrones.
- Los errores de ortografía se podrían optimizar usando una librería que calcule aproximaciones de texto, no solo resultados exactos.
- Los patrones de comparaciones numéricas, se podría crear un modelo de transformación de números en palabras, a su representación numérica, puesto

que en este momento la única forma de comparar valores es introduciendo el valor exacto en números, y no es posible incluir expresiones como “30 mil”.

5. CONCLUSIONES

- Por medio de una arquitectura por capas y del apoyo de frameworks que utilizan mejores prácticas en ingeniería de software, es posible crear aplicaciones con alta reutilización de código, bajo acoplamiento, y alta cohesión de sus partes.

- El manejo de Hibernate, o de un ORM en general, facilita el acceso a datos e independiza las particularidades de los motores de base de datos. Adicionalmente, desde un punto de vista de desarrollo, nunca se pierde de foco el modelo objetual, lo que permite tener mayor consistencia y entendimiento del software.

- Las expresiones regulares en Java tienen un gran potencial que facilita el reconocimiento de palabras o frases complejas. Si se combina esta herramienta, con un proceso de transformación de expresiones en tiempo de ejecución, se puede lograr una alta flexibilidad al momento de identificar palabras o frases incluso coloquiales, y convertirlas en consultas de base de datos.

- Trabajar con Hibernate permite tener una flexibilidad significativa para utilizar diferentes fuentes de datos. El concepto de unidad de persistencia de JPA se puede abstraer para manejar diferentes tipos de fuente de datos, y la integración con Spring permite también adaptar el modelo para soportar, de manera mantenible, múltiples bases de datos sin perder la legibilidad de la aplicación.

- La creación de consultas complejas de base de datos se puede descomponer en la sumatoria de unas cláusulas que proporcionan por si solas solo una porción de la consulta final, pero que por medio de agrupaciones pueden convertirse en una

expresión completa. La flexibilidad del modelo se refleja en que cada palabra de una búsqueda puede contribuir con su porción del resultado final.

- La herencia de Hibernate permite extender el modelo para cualquier tipo de establecimiento, conservando el soporte de múltiples patrones que operan sobre la entidad padre (Place) y sus relaciones.

- El manejo de mapas en Java provee la velocidad necesaria para consultar datos en memoria en un tiempo mínimo. Utilizando la clase ConcurrentHashMap se logra además manejar concurrencia entre sesiones.

- La inyección de dependencias es una técnica que permite reducir el acoplamiento entre componentes de una gran manera. En los casos en que fue necesario cambiar la implementación interna de uno de los componentes el impacto en los demás elementos de la aplicación fue mínimo, y por tanto se logra una mayor facilidad de mantenimiento y de extensibilidad de la aplicación.

- La programación orientada a aspectos facilita ciertos aspectos del desarrollo de software, como la transaccionalidad. Por medio de anotaciones es posible crear código más limpio, separando elementos transversales a la aplicación.

BIBLIOGRAFÍA

Procesamiento del lenguaje natural para recuperar información. Tomado de http://www.monografias.com/trabajos81/procesamiento-lenguaje-natural-recuperar-informacion/procesamiento-lenguaje-natural-recuperar-informacion2.shtml. Página visitada en Agosto de 2011.

Natural Languague Processing in Textual Information Retrieval and Related Topics. Tomado de http://www.upf.edu/hipertextnet/en/numero-5/pln.html. Página visitada en Agosto de 2011.

Aplicaciones del procesamiento del lenguaje natural en la recuperación de información en español. Tomado de http://coleweb.dc.fi.udc.es/cole/library/ps/Vil2005a.pdf. Página visitada en Agosto de 2011.

Journal of Pattern Recognition Reseach. Tomado de http://www.jprr.org/index.php/jprr. Página visitada en Septiembre de 2011.

Spring Framework Reference Documentation. Tomado de http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/. Página visitada en Septiembre de 2011.

Hibernate – Relational persistence for Idiomatic Java. Tomado de http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/. Página visitada en Septiembre de 2011.

Spring + JPA + Hibernate. Tomado de http://www.gbcacm.org/sites/www.gbcacm.org/files/slides/4%20-%20Spring-JPA-Hibernate.pdf. Página visitada en Agosto de 2011.

Multi-tenant architecture Spring + Hibernate. Tomado de http://forum.springsource.org/showthread.php?92767-Multi-Tenant-Architecture-Spring-Hibernate. Página visitada en Agosto de 2011.

The spring series – Part 1: Introduction to the spring framework. Tomado de http://www.ibm.com/developerworks/web/library/wa-spring1/. Página visitada en Agosto de 2011.

Ontology development 101: A guide to créate your first ontology. Tomado de http://www-ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html . Página visitada en Octubre de 2011.