Property Search Plugin Architecture

Overview

The Property Search Plugin is designed with a three-layer architecture that integrates with Eliza's knowledge system while maintaining clean separation of concerns. This document explains the architecture, data flow, and integration points.

Architectural Layers

1. Plugin Layer (`index.ts`)

The topmost layer that handles user interaction and plugin initialization.

```
export const plugin: Plugin = {
    name: 'property-search',
    description: 'Search and manage property data',
    services: [],  // Services created in handler with runtime
    actions: [searchPropertiesAction]
};
```

Key responsibilities:

- Plugin registration and configuration
- Action definition and handling
- Natural language query processing
- Response formatting

2. Service Layer (`services/index.ts`)

The business logic layer that coordinates operations.

```
export class PropertyStorageService implements Service {
    private storage: PropertyStorage;
    private runtime: IAgentRuntime;

    constructor(storage: PropertyStorage, runtime: IAgentRuntime) {
        this.storage = storage;
        this.runtime = runtime;
    }

    async searchByFilters(filters: FilterGroup): Promise<SearchResult[]> {
        return this.storage.searchByFilters(filters);
    }
}
```

Key responsibilities:

- Business logic coordination
- Service interface definition
- Runtime and storage management
- Future transaction handling

3. Storage Layer (`storage/memory-storage.ts`)

The data access layer that integrates direct property access with Eliza's knowledge system.

```
export class MemoryPropertyStorage extends BasePropertyStorage {
    private properties: Map<string, PropertyData> = new Map();
    private runtime: IAgentRuntime;

    async searchByFilters(filters: FilterGroup): Promise<SearchResult[]> {
        // Create memory object for knowledge search
        const memory: Memory = {
            agentId: this.runtime.agentId,
            userId: this.runtime.agentId,
            roomId: this.runtime.agentId,
            content: {
                text: this.filtersToQuery(filters)
            }
        };

        // Get results from both systems
        const knowledgeItems = await knowledge.get(this.runtime, memory);
        const directResults = // ... direct property search

        return [...knowledgeResults, ...directResults];
    }
}
```

Key responsibilities:

- Direct property storage and retrieval
- Knowledge system integration
- Query transformation
- Result aggregation

Data Flow

1. Search Request Flow

```
User Query
   ↓
Plugin Handler
   ↓
```

```
Create Storage & Service (with runtime)
   ↓
Service.searchByFilters()
   ↓
Storage Layer Processing
   ├→ Convert Filters to Query
   ├→ knowledge.get() Search
   └→ Direct Property Search
   ↓
Aggregate Results
   ↓
Format Response
   ↓
Return to User
```

2. Filter Processing

```
FilterGroup
   ↓
Convert to Text Query
   ├→ Field:Value pairs
   └→ AND/OR operators
   ↓
Create Memory Object
   ↓
knowledge.get()
```

Integration Points

1. Eliza Knowledge System

- Integration happens at the storage layer
- Filters converted to natural language queries
- Results merged with direct property search
- Metadata preserved through conversion

2. Runtime Integration

```
// Creation in handler
const storage = new MemoryPropertyStorage(runtime);
const service = new PropertyStorageService(storage, runtime);
```

- Runtime passed through all layers
- Enables access to Eliza's core systems
- Maintains plugin context

Type System

1. Core Types

```
interface FilterGroup {
    operator: 'AND' | 'OR';
    filters: (MetadataFilter | FilterGroup)[];
}

interface MetadataFilter {
    field: string;
    operator: FilterOperator;
    value: any;
}

type FilterOperator =
    | '$eq' | '$ne'
    | '$gt' | '$gte'
    | '$lt' | '$lte'
    | '$in' | '$nin'
    | '$exists'
    | '$near';
```

2. Result Types

```
interface SearchResult {
    id: string;
    property: PropertyData;
    similarity: number;
    matchedFilters: string[];
}
```

Future Enhancements

1. Query Enhancement

- Improved filter to query conversion
- Natural language understanding
- Context awareness

2. Result Merging

- Smarter result deduplication
- Relevance scoring
- Result ranking

3. Caching

- Query result caching
- Knowledge cache integration

- Cache invalidation

Best Practices

1. **Layer Separation**
    - Keep layers loosely coupled
    - Use interfaces for communication
    - Maintain single responsibility
2. **Error Handling**
    - Use custom error types
    - Proper error propagation
    - Meaningful error messages
3. **Runtime Management**
    - Pass runtime through constructors
    - Initialize services with runtime
    - Maintain runtime context
4. **Query Processing**
    - Clean input data
    - Handle edge cases
    - Validate filters

Testing Strategy

1. **Unit Tests**
    - Test each layer independently
    - Mock dependencies
    - Test error cases
2. **Integration Tests**
    - Test layer interactions
    - Test knowledge integration
    - Test full query flow
3. **E2E Tests**
    - Test user scenarios
    - Test with real runtime
    - Test full plugin flow