CloudConnect - Cloud Resource Manager

A simplified cloud management console that lets developers create, configure, and manage cloud resources similar to Azure services.

Overview

CloudConnect provides a modular, extensible system for managing three types of cloud resources: AppService, StorageAccount, and CacheDB. Each resource has its own configuration and lifecycle (create → start → stop → delete).

Features

- **Multiple Resource Types**: Support for AppService, StorageAccount, and CacheDB with unique configurations
- **Resource Lifecycle Management**: Create, start, stop, and delete resources with proper state validation
- **Activity Logging**: All operations are logged to individual resource log files
- **Extensible Architecture**: Easy to add new resource types using a registry-based system
- **CLI Interface**: User-friendly menu-driven command-line interface

Project Structure

```
.
├── main.py          # Main application and CLI interface
├── resources.py     # Resource classes and registry system
├── logger.py        # Logging functionality
└── logs/            # Directory for resource log files (created automatically)
```

Resource Types

AppService

Configuration includes runtime, region, and replica count.

- **Runtime**: python, nodejs, or dotnet
- **Region**: EastUS, WestEurope, or CentralIndia
- **Replica Count**: 1, 2, or 3

StorageAccount

Configuration includes encryption settings, access key, and storage capacity.

- **Encryption**: Enabled or disabled

- **Access Key**: Authentication key for access
- **Max Size**: Storage capacity in GB

CacheDB

Configuration includes TTL, capacity, and eviction policy.

- **TTL**: Time-to-live in seconds
- **Capacity**: Storage capacity in MB
- **Eviction Policy**: LRU, FIFO, or other strategies

Installation & Usage

Prerequisites

- Python 3.7+

Running CloudConnect

```
bash

python main.py
```

Menu Options

1. **Create Resource**: Select a resource type and configure it with the required parameters
2. **Start Resource**: Activate a stopped resource
3. **Stop Resource**: Deactivate a running resource
4. **Delete Resource**: Soft-delete a resource (marks it as deleted but keeps records)
5. **View Logs**: Display activity logs for a resource
6. **Exit**: Close the application

Example Usage

```
Enter choice: 1
Select resource type:
1. AppService
2. StorageAccount
3. CacheDB
Choice: 1

Enter resource name: appservice-1
Select runtime (python / nodejs / dotnet): python
Select region (EastUS / WestEurope / CentralIndia): WestEurope
```

```
Select replica count (1 / 2 / 3): 2
AppService created successfully!

Enter choice: 2
Enter resource name: appservice-1
AppService started at 10:42 AM in WestEurope
(Log written to logs/appservice-1.log)
```

Resource Lifecycle

Resources follow a state-based lifecycle:

1. **Created**: Resource is initialized in a "stopped" state
2. **Started**: Resource transitions to "running" state
3. **Stopped**: Resource returns to "stopped" state
4. **Deleted**: Resource is soft-deleted (marked as deleted but persists in logs)

Constraints

- Resources must be stopped before deletion
- Deleted resources cannot be started or stopped
- Duplicate resource names are not allowed
- Invalid operations produce clear error messages

Logging

All resource operations are logged to individual files in the `logs/` directory:

- Log files are named: `{resource_name}.log`
- Log entries include timestamps and operation details
- Example: `[10:42 AM] AppService started in WestEurope`

Extending CloudConnect

To add a new resource type, follow these steps:

1. Create a new class that inherits from `Resource` in `resources.py`
2. Implement the `get_details()` method
3. Decorate the class with `@ResourceRegistry.register("ResourceTypeName")`
4. Add configuration prompts in `create_resource()` method in `main.py`

Example:

```python
@ResourceRegistry.register("Database")
class Database(Resource):
    def __init__(self, name: str, engine: str, storage_gb: int):
        super().__init__(name)
        self.engine = engine
        self.storage_gb = storage_gb

    def get_details(self) -> str:
        return f"Database: engine={self.engine}, storage={self.storage_gb}GB"
```

Error Handling

CloudConnect handles various error scenarios gracefully:

- **Duplicate Resource Names**: Prevents creation of resources with the same name
- **Invalid Lifecycle Operations**: Prevents invalid state transitions (e.g., starting an already running resource)
- **Permission Errors**: Prevents operations on deleted resources
- **Input Validation**: Validates user selections and inputs

Architecture

Resource Registry Pattern

CloudConnect uses a registry pattern to manage resource types dynamically. This allows new resource types to self-register without modifying core logic.

Soft Delete

Resources are soft-deleted, meaning their metadata is retained in logs for record-keeping while they cannot be accessed or modified.