The KeyManager

One common problem when working with remote services is key management. The service providers want to be able to protect or monetize their service. Restricting access to authorized users requires a mechanism for recognizing who is authorized and who is not. An API key or some type of token is a very common way of controlling access. This means that developers have to manage which keys belong to which services and provide the right key to each service provider.

Before a developer can use an API with controlled access, the developer needs to register with the service provider and request a key. Service providers all have different ways of requesting a key. You will need to request keys before you can use the example notebooks.

API keys can be valuable - especially when use of the service results in charges. As a result, keeping keys secret is important. Many programmers will create an environment variable to hold each API key they will use. This strategy keeps the API key from being visible in the code. Instead, a few lines of code are used to extract the API key from the environment variable when it is needed.

In this approach, and just about any approach, the value of the API key must be stored somewhere, often in a file. For the approach that relies on setting an environment variable, there is a shell startup file or a configuration file that is read when the programming environment is launched.

This appendix introduces the KeyManager class and explains how to add keys and how to access keys in python code.

Interactive Shell: A Brief Tutorial

The KeyManager is a class that defines a set of operations on a data structure and can both save and restore the data in a file. The KeyManager class is in a stand-alone user module called `apikeys`.

Once you have downloaded and installed this python user module in your environment you need to import it into a Jupyter notebook. This looks like any other import statement:

```
from apikeys.KeyManager import KeyManager
```

Starting with Homework 2, the KeyManager class is used to store and retrieve API keys in the example code that shows how to use an API.

You can add new keys to the key manager programmatically, but the class file provides a simple interactive shell that you can run from a terminal or command line session. The interactive shell allows you to create new key entries, update data associated with the key, and expire keys.

The example below starts by adding keys from TMDB (TheMovieDB.org). It serves as a good example because TMDB has two different keys and that helps illustrate features of the interactive shell. However, the first key you probably want to add to the KeyManager is your key for the ORES API (to solve Homework 2).

You would run the interactive shell by executing the KeyManager source code file as a program. In a terminal session you would need to change directory to the appropriate subdirectory in the class library and type something like:

```
python KeyManager.py
```

That command launches the interactive shell and probably gives a warning that it could not open the key file. It will then provide the shell prompt 'Key manager >'. The example below shows the likely warning message, and the use of the question mark '?' command to get help.

```
The key file was missing.
Are you initializing a new key file?

KeyManager > ?

This interactive key manager provides rudimentary access to the
KeyManager functionality. All commands should be lowercase.

?, h, help
   Get this help message.
l, list
   List or print a brief version of key information.
n, new
   Interactive creation of a new key.
find d = <domain_name> u = <username>
   Find a specific key record. This will print the full record.
a, active
   Show the full information of the active record.
s, set <field> = <value>
   Create and/or set the value of an arbitrary field in the record.
expire
   Expire the active record. Expire the key of the active record.
q, quit
```

```
    Quit this interactive shell.
```

The help lists the commands that are recognized by the shell. The 'find' and 'set' commands take additional structured parameters.

The 'new' command creates a new key entry. This is an interactive process where you enter several pieces of information. This might look like the following for an API like TheMovieDB.org (TMDB):

```
KeyManager > new

To create a new key you need to enter three pieces of
information: domain name for the API, username, and key.
You can also add an optional description.

KeyManager      ['domain']> api.themoviedb.org
KeyManager    ['username']> rebert
KeyManager         ['key']> 0AF3-220C-FFB8-104D
KeyManager ['description']> API Key for The Movie DB (TMDB)
https://www.themoviedb.org
Creation was attempted. Check by listing key records.
Use the 'set' command to set additional or other optional fields.
```

TMDB is an interesting example because that API has two different types of keys for two different types of access. The creation of the key record above only allows the entry of one key. It could be possible to manage two different keys with two different key records, but simply putting both keys in the same record might be more efficient. The set command allows us to do that. However, before we can operate on a key, we need to make it the active key with the find command.

```
KeyManager > find u = rebert d = api.themoviedb.org
Found 1 record that met the conditions.
Working with the active record:
{
    "key": "0AF3-220C-FFB8-104D",
    "domain": "api.themoviedb.org",
    "username": "rebert",
    "organization": "",
    "mnemonic": "",
    "description": "API Key for The Movie DB (TMDB)
https://www.themoviedb.org",
    "created_ts": "2024-05-16 16:44:49",
```

```
    "updated_ts": "2024-05-16 16:44:49",
    "expired": false
}
```

This makes the TMDB key record the active record and we can now use the set command to set an additional field.

```
KeyManager > set access_token = 0AF3-011E-C0A0-74C0
Update to the record was attempted. Check by listing key records.
```

The set command defined an additional field in the record called 'access_token' and set it to the value specified in the command. We can check the values in the active record by using the 'active' command.

```
KeyManager > active
The current active record is:
{
    "key": "0AF3-220C-FFB8-104D",
    "domain": "api.themoviedb.org",
    "username": "rebert",
    "organization": "",
    "mnemonic": "",
    "description": "API Key for The Movie DB (TMDB)
https://www.themoviedb.org",
    "created_ts": "2024-05-16 16:44:49",
    "updated_ts": "2024-05-16 16:44:49",
    "expired": false,
    "access_token": "0AF3-011E-C0A0-74C0"
}
```

You can see which keys are being managed by the KeyManager with the 'list' command. Since we just put one key into the manager, a list should just show one key.

```
KeyManager > list

Here are the keys in the key file:

ACTIVE | USERNAME                  | DOMAIN
-------------------------------------------------------------
True   | rebert                    | api.themoviedb.org
```

```
KeyManager > quit
```

The list command shows the information necessary to perform a find command and make a specific key the active key.

The 'quit' command or an empty line will quit the interactive shell.

Accessing Keys in Code

One of the main reasons to have a KeyManager class is to simplify, or at least standardize, access to those keys in code. The introduction to the interactive shell, above, suggests some of the things that the KeyManager will do. For example, the find command allows keys to be organized by the domain of the API and the username who is authorized to use the API. Accessing the keys through python code uses that same general idea.

The KeyManager is like any class, it needs to be imported before it can be instantiated. That import statement should be something like:

```
from apikeys.KeyManager import KeyManager
```

That makes the class available in the code. Where the code needs access to the key, it instantiates the key, finds the key record, and then extracts the value of the key from the resulting dictionary. Those steps look like the following:

```
#   Instantiate the KeyManager
key_manager = KeyManager()
#
#   Returns a list of keys
key_list = key_manager.findRecord(domain="api.wikimedia.org")
#
#   Assume the first match is correct, get the key field
api_key = key_list[0]['key']
```

There are a few details that may not be obvious from the sample code. The first is that the findRecord() method can perform the search by domain name or by username or using both. The method always returns a list of matching records, or an empty list if no records match. By default the method returns active records. If you want to find expired records a parameter will allow you to find those. That is probably a special case, and you can review the KeyManager code to see how that is done.