
University of Glasgow | School of Computing Science

# Package Recommendation Engine

Keir Alexander Smith

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 15, 2015

**Abstract**

This paper covers the construction of a basic recommendation engine for operating system packages, specifically for use with the DNF package manager. The tool should allow users to discover useful packages through an easy to use command line interface.

# Contents


**1 Introduction** **1**

1.1 Problem overview . . . 1

1.2 Motivation . . . 1

1.3 Aims . . . 1

1.4 Report outline . . . 1

**2 Background** **2**

2.1 Modern Graphical Package Managers . . . 2

2.2 NUGET Concierge . . . 2

2.3 DNF . . . 2

**3 Design** **3**

3.1 Recommend Plugin . . . 3

3.2 Database . . . 4

3.2.1 Selection . . . 4

3.2.2 Internal Structure . . . 4

**4 Implementation** **6**

4.1 First Pass Mock UI . . . 6

4.2 Testing . . . 6

4.3 Back End Implementation . . . 6

4.4 Bringing it together . . . 7

**5 Evaluation** **10**

5.1 Survey Response . . . . . 10

**6 Conclusion** **11**

6.1 Summary . . . . . 11

6.2 Future Work . . . . . 11

6.2.1 Package Groups . . . . . 11

6.2.2 Weighted Relationships . . . . . 11

6.3 Lessons Learned . . . . . 12

**Appendices** **13**

**A Name of the first appendix** **14**

# Chapter 1

# Introduction

## 1.1 Problem overview

In a modern operating system there exists many packages for end users to install and this collection grows every day. Finding useful packages to install can be a laborious task, often involving the use of the internet to track down the package the user has been looking for, if it exists.
Furthermore there exists little support for installing packages commonly installed side by side. For example a user who has vim installed also installs Java Development Kit (JDK), the user may not be aware of the existence of a Java plugin for vim which is extremely useful.

## 1.2 Motivation

Package management is an interesting and very useful tool for many users, however the basic implementation has been static for many years. With the addition of this tool, we could see a decrease in users having to use search engines to look for packages they should be able to discover easily on command line.

## 1.3 Aims

This project aims to attempt to address the issues discussed above. Fore-mostly the problem of finding new packages by offering a powerful recommendation system for users to discover packages.
Consider a user on a Linux machine, looking for any new useful developer tools to help their work flow. Using an internet search engine returns a fairly miss match set of results, this project aims to supply that user with a command line interface where they can ask for a recommendation based on package of their choice. In this case our user asks for a recommendation based off of the JDK and is returned with a list of useful debugging tools and plugins which they weren't aware of.

## 1.4 Report outline

This report will begin by looking at the research undertaken at the outset of the project, then continue into design of the system. This will lead into the implementation and finally the evaluation and conclusion.

# Chapter 2

# Background

A package manager allows users to search for, install and update packages containing useful programs. For many years Unix has relied on package managers to allow easy management of tools and underlying applications. However in recent times, as package numbers increase and the ease of search engines becomes more prominent, searching using a command line tool has become less prevalent. Unless a user knows exactly what they want, often times they will resort to a internet search engine to find new packages.

## 2.1 Modern Graphical Package Managers

A more modern solution to this problem is the use of Graphical User Interfaces (GUIs) to abstract the annoyance of searching on command line away from the user. However this requires that the user is running a system with graphical output, a luxury which is often not found when running Virtual Machines (VMs) or using Secure Shell (SSH).

## 2.2 NUGET Concierge

In a similar vein NUGET (a .NET package manager) experimented with recommendations based off the same method this project uses, with weighting between packages rather than between user's install habits. A user could upload their project's meta data and Concierge would list packages the user may be interested in using based of what their project needs.

## 2.3 DNF

This project aims to supply similar functionality to users of DNF, Fedora 21's new package manager. DNF allows plugins to be easily added by simply dropping a Python file into the plugins directory. This allows DNF to be extended easily with little hardship from the user. Building this functionality into DNF is exactly the behaviour this project aims to provide to the end user.

# Chapter 3

# Design

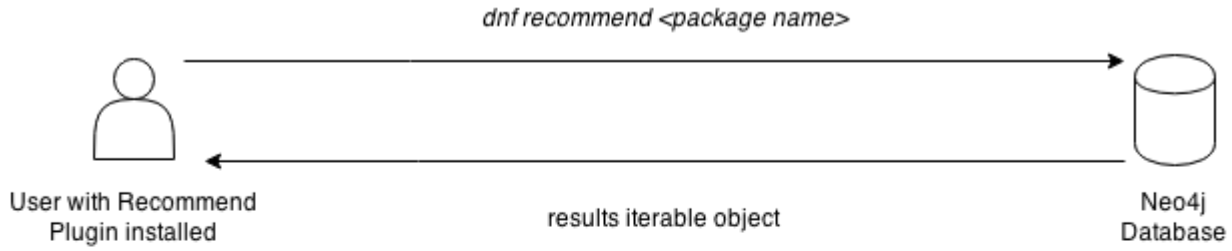
This system comes in two parts, a client side plugin for DNF which the user installs by dropping a single Python script into the correct directory. Also a graph server side database to store user's installed packages anonymously and provide data for recommendations.
Each of these components will be discussed in their own sections.

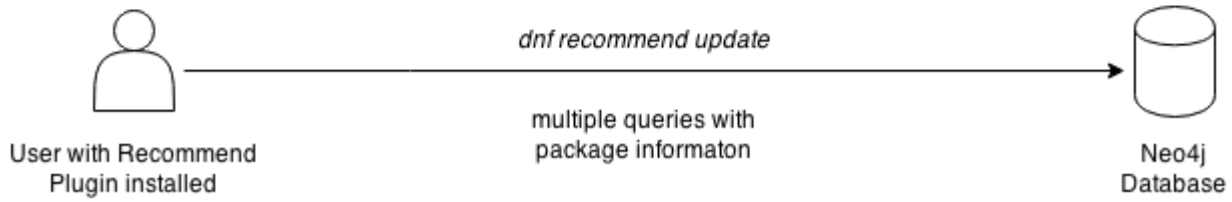
## 3.1 Recommend Plugin

In the choice between DNF and dpkg/apt, DNF was chosen due to it's excellent plugin support over the alternative.
The Recommend plugin has two key features:

- Request recommendation from server
- Upload user's installed packages anonymously

Both functions require a connection to the back end database, a connection over the internet is assumed.
The plugin needed to be easy to use, to counter the desire to open a browser. With this in mind it is designed with two clear commands.




## 3.2 Database

The project requires a back end data storage median. Immediately a database comes to mind for a long term, concurrent access method of storing and returning large volumes of data. Below the various potential choices of database are discussed.

### 3.2.1 Selection

Neo4j was selected as the database of choice for this project. There are several key reasons for this:

- NOSQL database which is built for finding paths in mind
- A standard SQL database would make storing the data complex
- Neo4j is built to scale across distributed systems being a modern database solution

### 3.2.2 Internal Structure

Internally the structure of the graph can be seen in the figure below.


One user may have many installed packages
One package may have been installed by many users
User
Installed
Package
Users are unique
Packages are unique

# Chapter 4

# Implementation

In regards to development environment, all work took place using a GitHub repository with all the Python scripts kept up to date there. Locally the Recommend plugin was kept in the Git repository folder which was then sym linked to the DNF plugin folder. This enabled work and change tracking to take place without disturbing work flow to test the plugin.
With the design settled, work began with a 'Wizard of Oz' style mock up, where the user could use the command line interface as if the system were complete, however anything they go back was simply place holder.

## 4.1 First Pass Mock UI

For this initial development two features needed to be implemented. A command to push installed packages to server and request a recommendation from the server.
DNF allows a developer to hook into its core functionality by extending classes, which allows new commands to be written and functionality added in a single Python script. Figure 4.1 and 4.2 shows two classes extending DNF plugin and command classes respectively.

## 4.2 Testing

In order to test this system, two scripts were written in Python to fill the database with both real and mock data. Several small Cypher scripts were also written to test that the database was working internally.
Listing 4.3 and 4.4 show the Python scripts for entering data.

## 4.3 Back End Implementation

Now that some form of client side had been written, the back end could be implemented. This turned into a simple case of downloading and running a Neo4j instance on a local machine to test scripts.
The initial script written, shown in figure, read a list of packages from a text file in the format which DNF dumps them and then created each on the database and tied them to a user with a fake ID.
With that was in place, scripts to find sets of recommendations between packages can be written and tested.

```
class Recommend(dnf.Plugin):

    name = 'recommend'

    def __init__(self, base, cli):
        self.base = base
        self.cli = cli
        global THIS_UUID
        try:
            id_file = open(HOME + "/.recommend_uuid", 'r')
            THIS_UUID = uuid.UUID(id_file.readline())
            id_file.close()
            print "UUID Successfully Read"
        except:
            print "No UUID File. New UUID"

        if self.cli is not None:
            self.cli.register_command(RecommendCommand)

    def get_installed(self):
        query = self.base.sack.query()
        installed = query.installed()
        return list(installed)
```

Listing 4.1: Recommend Class

## 4.4 Bringing it together

With the client side UI already written and tested, it was a simple case of changing a constant defining the server address to the operational database which was running on local host.

```
class RecommendCommand(dnf.cli.Command):
    aliases = ['recommend']
    summary = 'Makes a recommendation based on your currently installed packaged'

    def get_recommend_list(self, package):
        graph = Graph()

        search_term = package
        limit = "2500"
        search_string = 'MATCH n--u WHERE u.name = "' + search_term + '" AND n.id = "' + THIS_UUID.hex
             + '" MATCH u--n1 WHERE n1.id <> n.id MATCH n1--u1 WHERE u1.name <> "' + search_term + '"
            AND NOT u1--n RETURN u1 LIMIT ' + limit

        result = graph.cypher.execute(search_string)

        popular_counter = collections.Counter(result)

        print popular_counter.most_common(10)

    def make_graph(self, packagelist):
        graph = Graph()

        user_list = graph.merge("User", "id", THIS_UUID.hex)

        id_file = open(HOME + "/.recommend_uuid", 'w+')
        id_file.write(THIS_UUID.hex)
        id_file.close()

        for user in user_list:
            this_pc = user

        for package in packagelist:
            nodes = graph.merge("Package", "name", package.name)
            for node in nodes:
                relationship = Relationship(this_pc, "INSTALLED", node)
                graph.create_unique(relationship)

    def run(self, extcmds):
        """Execute the command."""

        #placeholder for recommend list
        rec_list = [100]

        if len(extcmds) > 1:
            print "Invalid Argments for Recommend plugin"
        elif len(extcmds) == 0:
            print("Please add the name of a package or use the 'update' command")
        elif extcmds[0].lower() == "update":
            print("Discovering Installed Packages...")
            self.base.fill_sack()
            packages = self.base.sack.query()
            packages = packages.installed()
            print("Building Graph...")
            self.make_graph(list(packages))
            print("Complete!")
        elif len(extcmds) == 1:
            print("Querying Graph for " + extcmds[0])
            self.get_recommend_list(extcmds[0])
        else:
            print "Invalid Argments for Recommend plugin"
```

Listing 4.2: Recommend Command Class

```
from py2neo import Graph, Node, Relationship

list_file = open("installed.txt", "r")

packagelist = []

for line in list_file:
    packagelist = packagelist + [line.split('.')[0]]

list_file.close()

graph = Graph()

user_list = graph.merge("User", "id", 1)

for user in user_list:
 this_pc = user

for package in packagelist:
    nodes = graph.merge("Package", "name", package)
    for node in nodes:
        relationship = Relationship(this_pc, "INSTALLED", node)
        graph.create_unique(relationship)
```

Listing 4.3: Fills the graph with real data

```
from py2neo import Graph, Node, Relationship
import random as r

list_file = open("packages.txt", "r")

packagelist = []

for line in list_file:
    packagelist = packagelist + [line.split('.')[0]]

list_file.close()

packagelist = packagelist[3:]

graph = Graph()

cycle = int(raw_input("Number of users: "))

#tx = graph.cypher.begin()
for i in range(1, cycle + 1):
    print "Commiting packages for user " + str(i)
    user_list = graph.merge("User", "id", i)

    for user in user_list:
        this_pc = user

    for package in packagelist:
        if r.randint(0, 10) < 1:
            nodes = graph.merge("Package", "name", package)
            #tx.append('MERGE (:Package {name:"' + package + '"})')
            #tx.append('MATCH (u:User {id:' + str(i) + '}) MATCH (n:Package {name:"' + package + '"})
                 CREATE UNIQUE (u)-[:INSTALLED]-(n)')
            for node in nodes:
                relationship = Relationship(this_pc, "INSTALLED", node)
                graph.create_unique(relationship)
    #tx.process()
```

Listing 4.4: Fills the graph with mock data

# Chapter 5

# Evaluation

## 5.1 Survey Response

# Chapter 6

# Conclusion

## 6.1 Summary

## 6.2 Future Work

There exists a lot of scope for this project to expand. This report will discuss two interesting areas for potential future work.

### 6.2.1 Package Groups

As a potential future addition, commonly installed package groups could be identified on the back end, which would be pushed forward to end users, allowing them to quickly install a complete set of tools.
An example of this would be web developers, who commonly install at least the three main browsers[1] and any debugging/developer tools associated with them.[2]
This kind of powerful inferencing can be done using a graph database made to look for groups like this. If it's possible to then analyse the group and identify what it is, this would give users a lot of power when installing packages.

### 6.2.2 Weighted Relationships

Weighted relationships is something which NUGET Concierge used to make its recommendations. It's a one directional value assigned between two package nodes to determine how likely it is that one is installed on the same system with the other.
At it's core, it's very simple, however we could go lengths to make this weighting more meaningful, perhaps through the use of labels or special cases we can identify when best to include packages in a recommendation. For example it may be the case that package X is usually only installed when package Y is installed and Z isn't. A weighted relationship would not identify this as it would only recommend the most common. By tagging the relationships we can ensure the correct recommendation is made to the user.

[1] Google Chrome, Mozilla Firefox and Safari
[2] For example Firebug

## 6.3 Lessons Learned

# Appendices

# Appendix A

# Name of the first appendix