Extracting Dependency Tree from Common Package Managers

May 29, 2016

Scope

In this report we will present our dependency tree representation format as well as ways to extract it not only from the reference package manager implementation - NPM - but from others which have yet to be implemented. We will specify the required information as well as practical ways of extracting it from various package managers beside NPM. We will not, however, delve deeply into methods of reformatting, serializing or encoding the required information once it has been extracted.

Dependency Tree Data Structure

We've chosen to model our extracted dependency tree after the structure produced by NPM - the Node Package Manager. Dependencies are a list of objects, each containing the package name, resolved version, resolution constraints - or where the version selected was resolved from and, optionally, any subdepedencies. An example of such a tree, serialized to JSON is below:

```
{
  "name":"NodejsStarterApp",
  "version":"0.0.1",
  "dependencies":[
    {
      "name":"cfenv",
      "version":"1.0.3",
      "from":"cfenv@>=1.0.0 <1.1.0",
      "dependencies":[
        {
          "name":"js-yaml",
          "version":"3.4.6",
          "from":"js-yaml@>=3.4.0 <3.5.0"
        }
      ]
    },
    {
      "name":"marked",
      "version":"0.3.3",
      "from":"marked@0.3.3"
    }
  ],
  "platform":"NodeJS"
}
```

While similar to the representation returned natively by NPM, the exact format is not identical. The original format is presented more in depth in the next part.

NodeJS/NPM

NPM may produce the following output when queried for a project's dependency tree, serialized as JSON:

```
$ npm ls -json
{
  "name": "NodejsStarterApp",
  "version": "0.0.1",
  "dependencies": {
    "cfenv": {
      "version": "1.0.3",
      "from": "cfenv@>=1.0.0 <1.1.0",
      "resolved": "https://registry.npmjs.org/cfenv/-/cfenv-1.0.3.tgz",
      "dependencies": {
        "js-yaml": {
          "version": "3.4.6",
          "from": "js-yaml@>=3.4.0 <3.5.0",
          "resolved": "https://registry.npmjs.org/js-yaml/-/js-yaml-3.4.6.tgz"
        }
      }
    },
    "marked": {
      "version": "0.3.3",
      "from": "marked@0.3.3",
      "resolved": "https://registry.npmjs.org/marked/-/marked-0.3.3.tgz"
    }
  }
}
```

Specifically, dependencies are a map from the dependency name to an object that describes the resolved dependency - it's version, the constraints that lead to that version being chosen, the resolved package URL according to NPM and any sub-dependencies.

While similar, our data structure omits some extraneous information and reorganizes the dependency specification from a map to a list of objects that include the name of the package but omit the resolved URL. Conversion between the two formats should be trivial and was in fact implemented in our reference buildpack in only about a dozen lines of code.

Maven

Maven is a powerful build tool for Java projects that includes some dependency resolution and management capabilities. While Maven includes a powerful mechanism of resolving dependency version conflicts, unlike NPM it does not support specifying a dependency version range - only a specific version.

Once maven is installed, one can simply run the tree goal of the dependency plugin (which is built into maven itself). The resulting file will have content with the following format:

```
$ mvn dependency:tree -DoutputFile=dependencies.txt
$ cat dependencies.txt
omri:test:war:1.0-SNAPSHOT
\- org.springframework.ws:spring-ws-core:jar:2.1.4.RELEASE:compile
   +- org.springframework.ws:spring-xml:jar:2.1.4.RELEASE:compile
   +- org.springframework:spring-context:jar:3.2.4.RELEASE:compile
   |  \- org.springframework:spring-expression:jar:3.2.4.RELEASE:compile
   +- org.springframework:spring-aop:jar:3.2.4.RELEASE:compile
   |  \- aopalliance:aopalliance:jar:1.0:compile
   +- org.springframework:spring-oxm:jar:3.2.4.RELEASE:compile
   +- org.springframework:spring-web:jar:3.2.4.RELEASE:compile
   +- org.springframework:spring-webmvc:jar:3.2.4.RELEASE:compile
   +- wsdl4j:wsdl4j:jar:1.6.1:compile
   +- javax.xml.stream:stax-api:jar:1.0-2:compile
   +- commons-logging:commons-logging:jar:1.1.1:compile
   +- org.springframework:spring-core:jar:3.2.4.RELEASE:compile
   \- org.springframework:spring-beans:jar:3.2.4.RELEASE:compile
```

While not an easily parseable JSON string, this file contains all the necessary information to generate an object much like the one described above. Each line describes a unique package name, its resolved version and, optionally, any dependencies it may also have. Notably missing is information corresponding to the "from" string, describing the constraints that lead to this particular version being chosen. However, as we know, Maven does not support this capability in the first place, the from string is simply the package name followed by the resolved version.

Python/PIP

PIP is a popular package manager for Python. Unfortunately, Python packages are always installed globally, unlike NPM or Maven, that download and install packages locally, straight into the project directory. The de-facto solution to this is to use a Python utility called virtualenv that helps emulate separate Python installations per project, with their own version of pip and a private pip install space.

As pip packages are all installed globally (or in a simulated global environment,) the common way to list project dependencies is simply to list all packages installed using

```
pip freeze
```

Unfortunately, this returns a flat list of all packages installed and their currently installed version. This does not include any information about the dependency tree that lead to those packages being installed, nor the constraints that lead to a particular version being selected. However, there's a third-party Python utility that extracts the dependency tree, including the version constraints that lead to a version being selected for installation. The utility is itself installable via pip and is called pipdeptree. Running pipdeptree on a project that directly depends only on Flask, for example, might produce the following output:

```
$ pipdeptree
Flask==0.11
  - itsdangerous [required: >=0.21, installed: 0.24]
  - click [required: >=2.0, installed: 6.6]
  - Werkzeug [required: >=0.7, installed: 0.11.10]
  - Jinja2 [required: >=2.4, installed: 2.8]
    - MarkupSafe [installed: 0.23]
```

Pipdeptree also supports a more easily parseable JSON output that contains the exact same information and more, but the output is somewhat verbose so an example was withheld from this report for brevity. It should be quite easy to transform the JSON output of pipdeptree into the required format.