THE MOD

DANIEL C. BÜNZLI

Draft of January 15, 2008

1 INTRODUCTION

The mod is a proposal for a source level, distributed, localized and minimalistic package system to share OCaml modules among developers. What does this mean ?

Source level. The mod only manages packages of OCaml and C binding source code.

Distributed. There is no central repository of packages. Anyone who can publish on a web server can publish a package.

Localized. Packages are (at least conceptually) managed per project, not per machine. In a project the contents of a package can be installed, uninstalled, upgraded and downgraded by issuing a single command. Project dependencies become explicit.

Minimalistic. The architecture of the mod is simple and easy to understand, it puts the user in control. It is designed to solve the average use case.

Developers. The mod is for developers, it is not a mean to install software for the end user.

The goal of the mod is to facilitate module sharing and encourage the modularization and atomization of packages. A package containing a single module without any dependencies should not be seen as an anomaly however the effort spent to be able to use it or publish it should be proportional to its size. The mod seeks to minimize this effort.

Although it may provide hints and means for package standardization, the mod tries to be unintrusive with respect to the development practices of its users. The mod is about providing a minimal infrastructure to share modules.

2 ASSUMPTIONS

Running the mod depends only on the standard OCaml distribution. It relies entirely on `ocamlbuild` to build packages. Modular `ocamlbuild` plug-ins will be needed to simplify the use of packages that contain C bindings.

3 CONCEPTS

The mod is centered around two kinds of resources: packages and versions. A package is accessed via an URI like `http://example.org/example`. A version is accessed via a package. Packages and versions can be seen as dictionaries mapping keys to values.

A *package* defines a time ordered sequence of versions. It has *at least* two keys: *identifier* (pid) and *versions*.

- *Identifier* is a globally unique identifier. It should follow the reverse reverse domain name convention, e.g. `mypackage.example.com`. The identifier need not be related in any way to the URI used to access the package — packages may be accessed via multiple URIs.
- *Versions* is a time ordered list of versions of the package.

A *version* is a particular release of a package. It has *at least* two keys: *identifier* (vid) and *contents*.

- *Identifier* is an identifier unique in the package. It must have the form `yyyy-mm-dd [hh:mm:ss]` and denote the UTC time at which the version was published. The granularity of time for the identifier should only be as fine as to allow to unambiguously identify each version. The *latest version* of a package is the version with the latest time stamp. Using timestamps as version identifiers frees the developer eager to share from naming duties. Labels can be used to implement a custom versioning scheme (see below).
- *Contents* is a couple (uri, digest) made of an URI pointing to an archive and a cryptographic digest of it. The contents of the archive knows how to build itself with `ocamlbuild` (no other build system is supported).

A version may need a version of another package to provide its service. These dependencies are stored in the *dependencies* key of a version.

- *Dependencies* defines a list of versions on which the version depends. A *dependency* is a quadruplet (uri, pid, vid, digest) made of a package URI, a package identifier, a version identifier, and the digest of the version content. Two dependencies are equal if their three last components are.

A project can only depend on a *single* version of a package. A *conflict* occurs if there are two dependencies with the same package identifier but a different version identifier.

Both packages and versions have a key called *labels*. Labels are short identifiers to attach information to resources. In packages labels should make a quick description of it, for example camlzip could have the following labels, `zip`, `gzip`, `compress`, `libz`, `bindings`, `lgpl+static`. In versions, labels can be used to implement a versioning scheme in addition to the default one provided by timestamps. Version labels could be anything like `1.1.2`, `beta`, `stable`, `bugfix`, etc.

4 THE MOD FOR PACKAGE USERS

The general invocation scheme of the mod is :

```
> mod [cmd] [resource]
> mod [key] [resource]
```

The first form is used to act on the system while the second one is used to query resources. Prefix specification of commands, keys and resource identifiers is allowed as long as it is not ambiguous. For example `rem` for `remove` or `mypa` for `mypackage.example.com`.

The mod must be invoked in the directory where you invoke `ocamlbuild`. This will eventually create the `mod` directory at that location. It has the following layout :

- `mod/deps` file listing the dependencies of your project. This file becomes part of your build system and should be tracked by your revision control system.

- `mod/_files/` directory containing downloaded package files and extracted archives of version contents. The information therein may be updated and regenerated, it should be ignored by your revision control system. You can also safely remove this directory manually.

Once you have a hand on the location of a package you can add a dependency on it by invoking the command add that will add the *latest version* of the package as a dependency and recursively add its dependencies to the project.

```
> mod add http://example.org/example
Dependencies:
 mypackage.example.com
 2007-11-11 1.0
```

This downloads only the package file and those of its dependencies, not the actual version contents. The only way to explore packages is to add them as a dependency as above, this minimizes command complexity. A package and its dependencies can be easily removed with `remove`.

Now that we have the package dependency we can ask for its different versions by using the `versions` key.

```
> mod versions mypa
mypackage.example.com
Versions:
 2007-11-01 zero
 2007-11-11 1.0 stable
```

The dependencies of a particular version can be listed with the `dependencies` key. Unambiguous version labels can be used instead of version identifiers (in the example below `zero` instead of `2007-11-01`).

```
> mod dep mypa zero
mypackage.example.com 2007-11-01 zero
Dependencies:
 None.
```

To switch to the `zero` version of our package we just `add` the dependency. Since a project can only depend on a single version of a package this will remove the other dependency (there may be more steps involved if the version was installed or others depended on that version) and add the new one.

```
> mod add mypa zero
Dependencies:
 mypackage.example.com
 - 2007-11-11 1.0 stable
 + 2007-11-01 zero
```

To install the files of a dependency simply type:

```
> mod install mypack
```

This will download the archive of the version and extract it in the directory `mod/_files/mypackage.example.com/`. More complex packages may also go through a configuration step to locate system libraries (mainly for C stubs).

To use the package you will have to look the documentation in the package (if we impose more on packagers a new key could be introduced). However for pure OCaml modules it will usually be sufficient to add the following to your `ocamlbuild _tags` file :

```
<mod/_files/mypackage.example.com> : include
```

To check for new versions, package files need to be refreshed manually with the `refresh` command.

```
> mod refresh mypack
mypackage.example.com
No new version.
```

To be informed of new versions automatically you can put the URI location of the package in your news feed reader as package files are in fact just Atom news feeds [1].

If you want to remove the version's files you can type :

```
> mod uninstall mypack
```

This will remove the files of the version but it does not drop the dependency. You can quickly reinstall by issuing `install`.

The following command schemes are all you need to know to use the mod.

- `mod dependencies ([pid] [vid])*` lists the dependencies of the given package versions. If none is provided, lists all the dependencies of the project.
- `mod add [uri|pid vid]` adds or replaces the given package dependency. If [vid] is not present the latest version of the package is used.
- `mod remove [pid]*` removes the given dependencies. If [pid] is not present all dependencies are removed. If you want to change the version of a package do not remove it, use `add`.
- `mod refresh [pid]*` refreshes the given packages by downloading their package files.
- `mod upgrade [pid]*` refreshes the given packages and adds their latest version as a dependency. Upgrades every dependency if no pid is specified.
- `mod [key] [pid] [vid]` retrieves the value for the key of the given package or version.
- `mod install [pid]*` install the dependency on the given pids. Installs every dependency if no pid is specified.
- `mod uninstall [pid]*` uninstall the dependency on the given pids. Uninstall every dependency if no pid is specified.
- `mod clean` removes `mod/_files`.

5 THE MOD FOR PACKAGE PUBLISHERS

The system tries to make it easy for developers to publish packages and inform others that they were updated. In order to do so the mod uses Atom news feeds [1] as package files. One caveat is that feeds are written in XML and it is notoriously criminal to make humans edit plain XML.

For this reason a package is described in a raw UTF-8 encoded text file, *the package description file* which is converted by the mod to an Atom package file.

The description file can be seen as a simple tagged README or CHANGES file. Tags allow the specification of versions, release notes, dependencies, rights, authors, contributors, web pages for the package etc. The mapping between tags in description files and Atom XML elements can be found in section 6.

The following is an example of a minimal description file for a package `mypackage.example.org` authored by nobody. There are two versions in this package.

```
@id mypackage.example.org
@author nobody
@labels useless gpl

@version 2007-11-01
@labels  zero
@content http://www.example.org/mypackage/zero.tbz
         55e1beb09addacabfc35f1921913abb9

@version 2007-11-11
@labels  1.0 stable
@content http://www.example.org/mypackage/v1.0.tbz
         10f8ac0cdfd3eca081771d960382eb71
```

A description file can be converted to the actual package description format by using the package command.

```
> mod package README > /var/www/example/mypackage.atom
```

The following package is more complex. It mentions a project web page, versions have release notes, contributors and dependencies.

```
@id otherpackage.example.org
@author nobody @email nobody@@example.org
@webpage http://example.org/otherpackage

@version 2007-11-02
@notes First public release.
@content http://www.example.org/other1.0.0.tbz
         55e1beb09addacabfc35f1921913abb9

@version 2007-11-03
@labels 1.0.1
@notes Bugfix release. The package now depends on mypackage.
@contributor Mr. Bugfix @email bugfixer@@example.com
@dep
  http://www.example.org/mypackage
  mypackage.example.org
  2007-11-11 10f8ac0cdfd3eca081771d960382eb71
@content http://www.example.org/other-2007-11-11.tbz
```

```
          10f8ac0cdfd3eca081771d960382eb71
```

If you used the mod to develop your package, dependencies for the version you are about to release can be output as `@dep` tags with `mod dependencies -pack`.

The version archive can be created manually or if you list its contents in the file `mod/manifest` — to be tracked by your revision control system — by issuing the `archive` command. This will write the archive content to a file and print back its cryptographic hash.

```
mod archive /var/www/example/other-2007-11-11.tbz
Digest:
 10f8ac0cdfd3eca081771d960382eb71
```

The archive should contain the appropriate `_tag` file and `ocamlbuild` plug-in needed to build the package.

TODO. More work is needed to streamline the package publication procedure. E.g. a command line interactive mode to update the tagged README with a new version, automatically computing the version identifier (via time), dependencies (via `mod/deps`), version contents (via `mod/manifest`) and publish both the new news feed and the new version content to a given directory. In any case the README can always be reedited by hand.

6 PACKAGE FILE REFERENCE

In this section we describe how package description files are mapped to an Atom file [1] representing a package file.

The essence of the mapping is that the `atom:feed` element represents the package and each `atom:entry` a version of the package. Atom files contain all the semantic structure needed to describe release notes, authors, rights, link to a site, labels, etc. The only extensions that needed are to describe dependencies in `atom:link` elements:

1. A new value for the `rel` attribute of `atom:link`, `dependency`.
2. Three new `atom:link` attributes to identify versions, `mod:pid`, `mod:vid`, `mod:digest`. Atom explicitly supports such extensions via XML name spaces [2].

The whole package file is wrapped into an `atom:feed` element containing the following elements.

- `@id ID`
  Mandatory. The package id.

  ```
  <atom:id>id:ID</atom:id>
  ```

- `@author NAME [@email EMAIL] [@uri URI]`
  Mandatory (at *least* one). The person(s) currently responsible for the package.

```
<atom:author>
 <atom:name>NAME</atom:name>
 <atom:email>EMAIL</atom:email>
 <atom:uri>URI</atom:uri>
</atom:author>
```

- `@contributor NAME [@email EMAIL] [@uri URI]`
  Contributor(s) to package.

  ```
  <atom:contributor>
   <atom:name>NAME</atom:name>
   <atom:email>EMAIL</atom:email>
   <atom:uri>URI</atom:uri>
  </atom:contributor>
  ```

- `@labels ID*`
  Labels describing the package.

  ```
  <atom:category term="ID"/>...
  ```

- `@rights TEXT`
  Copyright information for the package (not the license itself, the license should be in the version's archive).

  ```
  <atom:rights>TEXT</atom:rights>
  ```

- `@webpage URI`
  Webpage for the package.

  ```
  <atom:link href="URI" rel="related"/>
  ```

- `@icon URI`
  Icon for the package.

  ```
  <atom:icon>URI</atom:icon>
  ```

- `@logo URI`
  Logo for the package.

  ```
  <atom:logo>URI</atom:logo>
  ```

Following these elements there may be a number of versions each wrapped into an `atom:entry` element and described with the following tags.

- `@version yyyy-mm-dd [hh:mm:ss]`
  Mandatory. The version id (and publication date of the package). No two version ids should be identical in the same package.

  ```
  <atom:id>id:yyyy-mm-ddThh::mm:ssZ</atom:id>
  <atom:updated>yyyy-mm-ddThh::mm:ssZ</atom:updated>
  ```

- `@labels ID*`
  `@author NAME [@email EMAIL] [@uri URI]`
  `@contributor NAME [@email EMAIL] [@uri URI]`
  Respectively the version's labels, authors (if different from the package) and contributors. Mapping is the same as given above for packages.

- `@notes TEXT`
  The version's release notes.

  ```
  <atom:content type="text">TEXT</atom:content>
  ```

- `@content URI HASH`
  The link to the archive and its hash.

  ```
  <atom:link href="URI" rel="enclosure" mod:digest="HASH"/>
  ```

- `@dep URI PID VID HASH`
  Dependencies of the package.

  ```
  <atom:link href="URI" rel="dependency"
             mod:pid="PID" mod:vid="VID" mod:digest="HASH"/>
  ```

7 TODO

- Package description files. Allow the specification of relative links and alternate download locations.
- Conflict management. Need a way to locally override a dependency in case two packages depend on two different version of the same package.
- Need a way to integrate documentation. So that you can easily get to the ocamldoc documentation of your packages.
- Any textual information is UTF-8 encoded. Package ids and file names should be restricted to ASCII letters only for file system compatibility.
- Use `atom:right` ? Or prefer a LICENSE file in the package ? At least labels should give a hint of the license.
  See also `http://www.rfc-editor.org/rfc/rfc4946.txt`.
- Resource consumption. On unices per user `.mod` cache with hard links.

BIBLIOGRAPHY

[1] M.Notthingham, R. Sayre. *The Atom Syndication Format*. RFC 4287, 2005.
`http://tools.ietf.org/html/rfc4287`

[2] Tim Bray et al. *Namespaces in XML 1.0 (2nd edition)*. W3C recommandation, 2006.
`http://www.w3.org/TR/REC-xml-names/`