Scala Bazaars Manual

Alexander Spoon

June 2007

1 Overview

Scala Bazaars supports Scala enthusiasts in sharing the software they create with each other. The community is strengthened when people can help each other out by sharing code. A programming language needs an archive of library code to be its most useful.

Scala Bazaars has several features specialized for this kind of community:

- Since programming efforts in the community are distributed around the world, the system must allow loose, decoupled collaboration. Participants are not required to wait for each other nor for any centralized authority.
- Since subgroups of the community have their own needs—just consider groups doing commercial development!—the system should support subgroup-specific access control policies.
- Since many open source projects are updated frequently, the system must allow conveniently updating components as new versions become available.
- Since, even in the common open-source group, there are different tolerances for stable versus new offerings, the system supports having multiple public servers with different update policies

Scala Bazaars is most closely related to Debian's APT, though it shares spirit with systems including: YUM, FreeBSD ports, CPAN, CTAN, SqueakMap, and Fink. The main difference from APT is that Bazaars tries to make it not only possible, but convenient, for subgroups to run their own servers with their own access policies.

The present document is a reference manual for Scala Bazaars. It is complete at the expense of readable. It does include full specifications for all components of the system, including the command-line interface, all file formats, and the network protocol. Those wanting to use the system should first check whether a tutorial exists for the specific need. Those wanting a more philosophical look at the system, including comparisons to other systems and to the literature, should look at the Package Universes Architecture document. It is available at the Scala Bazaars home page:

```
http://lamp.epfl.ch/~spoon/sbaz/
```

2 Architectural Concepts

This section describes the architecture as a whole. Each component of the architecture is described in detail in a later section.

Users of the system have a *local managed directory* that includes content supplied by the system. In general, users should not modify content in a managed directory except via a Scala Bazaars

system tool. The one exception is the `config` directory, which is explicitly intended for user modification and should not be modified by the Scala Bazaars tool.

Sharable content is held by *packages*. A package can be installed into a local managed directory, in which case all of its included files are extracted into the managed directory. A package can also be removed from a local managed directory, causing all of the files it installed to be removed.

Each package has a name and a version. Packages with the same name may be substituted for each other, but packages with larger version numbers are preferable in some sense.

A *bazaar* is an evolving set of packages. Each managed directory is associated with one bazaar. The system makes it convenient to choose and install packages from a managed directory's own bazaar, but not from other bazaars. The set of available packages evolves as the community associated with the bazaar shares and retracts packages over time.

Packages may depend on other packages. All dependencies are resolved within a single bazaar. The system does not allow installing or removing packages in a way that a managed directory's dependencies become unsatisfied.

On the whole, this approach gives convenient upgrades (simply grab the newest version of everything), loose coupling (you can wait for a while to upgrade), and convenient dependency management (requesting the install of a package automatically installs the packages it depends on). The cost of this approach is that packages must be posted separately to each bazaar they are useful in, but as argued in the architecture document, that cost appears to be ephemeral.

Bazaars may be combined in a variety of ways. Each bazaar has its own access policy, and thus combinations of bazaars with different access policies can yield a wide variety of useful configurations.

Bazaars do not hold the packages themselves. Instead, a bazaar holds package advertisements, and the packages themselves must be posted separately. Each package advertisement includes a URL referencing the associated package.

3 Bazaars and Their Descriptors

There are several kinds of bazaars that sbaz understands. Each of them is described below. To describe a particular bazaar, one uses a *descriptor*. There are XML descriptors for every legal bazaar, and their formats are defined alongside the definition of each kind of bazaar. Additionally, there is a short, non-XML form of descriptor given at the end of this section. The short form is more convenient for human editing, and is general enough to describe most bazaars that are used in practice.

3.1 Simple Bazaar

A simple bazaar includes the packages advertised on a single bazaar server. An example XML description of a simple bazaar is:

```
<simpleuniverse>
  <name>scala-dev</name>
  <location>http://scala-webapps.epfl.ch/sbaz/scala-dev</location>
</simpleuniverse>
```

The location tag specifies which Bazaars server supplies packages for this universe. The name is used by the command-line interface to Scala Bazaars whenever it is necessary to specify a specific simple bazaar within a compound bazaar.

3.2 Empty Bazaar

The empty bazaar holds no packages at all. Its XML description is simply:

```
<emptyuniverse/>
```

3.3 Override Bazaar

An override bazaar combines the packages from multiple other bazaars, with packages in later bazaars overriding packages in earlier ones. The overriding is name-based: a package in a later bazaar causes *any* same-named package in an earlier bazaar to disappear from the combined override bazaar.

An example XML description is as follows:

```
<overrideuniverse>
<components>
  <simpleuniverse>
    <name>scala-dev</name>
    <location>http://scbaztmp.lexspoon.org:8006/scala-dev</location>
  </simpleuniverse>
  <simpleuniverse>
    <name>local-hacks</name>
    <location>http://localhost/sbaz/local-hacks</location>
  </simpleuniverse>
</components>
</overrideuniverse>
```

3.4 Filter Bazaar (not yet implemented)

A filter bazaar includes those packages from some other bazaar whose name matches a designated regular expression. Filter bazaars are thus useful for taking just a few packages from an existing server.

3.5 Literal Bazaar (not yet implemented)

A literal bazaar includes a fixed set of packages. Literal bazaars are mainly useful for testing.

3.6 Short, Non-XML Descriptors

There is a short descriptor format available for describing many common bazaars. Each line of the format includes a name and a URL, thus designating a simple bazaar. If the descriptor has exactly one line, then that line designates the simple bazaar. for the whole descriptor. If the descriptor has multiple lines, then it designates an override bazaar combining the bazaars listed on each line. If the descriptor includes no lines, then it designates the empty bazaar. As an example, the following descriptor is equivalent to the previous example of an override bazaar:

```
scala-dev http://scbaztmp.lexspoon.org:8006/scala-dev
local-hacks http://localhost/sbaz/local-hacks
```

4 Access Control

Many organizations that use Scala Bazaars do not want to allow arbitrary accesses from unknown users. Scala Bazaars includes a simple access control system to limit usage as appropriate for the organization.

The approach is based on *keys*. A bazaar server has a list of keys that it considers valid. Each request to a server must either include a valid and sufficient key, or it must be in the subset of requests configured as *open access* for the particular server.

Each key is associated with a *message pattern* denoting the subset of requests it can authorize. The following message patterns are supported:

Read Requests that ask for the list of available packages recorded on a bazaar server. In XML, it looks like:

```
<read/>
```

Edit Requests that modify the list of available packages recorded on the server, including posting new advertisements, removing advertisements, and updating an advertisement in place. Edit request patterns include a regular expression that limits the request pattern to those requests involving a package whose name matches the regular expression. The semantics of the supplied regular expression are as for Java's regular-expression library. In XML, it looks like:

```
<edit nameregex="sbaz.*"/>
```

Key Edit Requests that modify the list of valid keys known to a server. There is only one request-pattern for key editing. Any client that knows a valid key-edit key can perform arbitrary manipulations on the set of keys known to a server (and thus transitively has full access to the server). In XML, a key edit pattern looks like:

```
<editkeys/>
```

A key includes three pieces of information: a request pattern, a short, descriptive text, and a string of decimal digits. The descriptive text is human-readable and allows keys to be associated with principles in an external access control system. Typical uses are names, email addresses, usernames, and employee ID's. The string of decimal digits is typically randomly generated and must be unguessable in order for the server to remain protected.

In XML, a key looks like:

```
<key>
  <messages>
    <edit nameregex="lex-.*"/>
  </messages>
  <description>lex@lexspoon.org</description>
  <data>264059714505207215086380670866232588803</data>
</key>
```

A group of keys can be stored in an *keyring*, which in XML looks like:

```
<keyring>
<key>
  <messages>
    <editkeys></editkeys>
  </messages>
```

```
  <description>keyservlet</description>
  <data>405971450520721508638067086623258803 26</data>
</key>
<key>
  <messages>
    <edit nameregex="sbaz-.*"></edit>
  </messages>
  <description>lex@lexspoon.org</description>
  <data>26405971450520721508638067086623258803</data>
</key>
<key>
  <messages>
    <editkeys></editkeys>
  </messages>
  <description>lamp</description>
  <data>059714505207215086380670866232588032 64</data>
</key>
</keyring>
```

The keys-based access control system of Scala Bazaars is both usable directly and supports, it is hoped, a variety of useful access policies. For simple access policies, the server administrator (or someone to whom they delegate) can manually manage and distribute keys. Note that key data can be transmitted via email and web sites, thus allowing distribution of keys to piggyback on existing secured infrastructure.

For organizations that have too many users for manual management of access, the keys approach supports writing a simple web server that can manage and distribute keys in an arbitrary fashion. For example, one could write a web server that gives out keys to people who have a valid login according to an LDAP server; nightly, the server could revoke keys for any users whose LDAP entry has disappeared.

5 Common Configurations

The available bazaar types and access-control policies described above can be combined to form a number of useful configurations. This section outlines several of them.

No restrictions Wikis have proven that effective and useful communities can be built even with no security restrictions at all. This policy can be implemented by configuring the server with all requests as open access.

Full access, but only to community members Many open-source projects have an organization of this form, including Debian's "Debian Developers" and FreeBSD's "committers." This policy can be implemented by giving a key-editing key to the membership gate keepers. The last step of admitting a new member to the community is to give them their own keys to manipulate the community bazaar servers.

Single provider, multiple users An individual developer wants to provide a suite of packages to be used by a larger community—possibly the world, or possibly a limited base of subscribers. To implement this policy, the developer can keep editing keys for personal use but publish the read key to the desired user base.

Moderation queues Sometimes it is desirable to have a large community contribute suggested packages, but a smaller group to decide on what is actually included. A moderation queue can be implemented as a separate bazaar where a full edit key (with regular expression ".*" is made available to whichever community is allowed to contribute suggestions (perhaps the entire world). Moderators would have both a read key for the moderation queue and a full edit key for the main bazaar, and could copy packages from the queue to the main bazaar whenever they are deemed appropriate.

Private local development Individual groups can form their own bazaar for private development without needing to coordinate with any central organization. They simply create the bazaar and share keys with members of the group according to their own local security policy.

Public libraries plus local development The above organization can be refined by allowing developers to use publicly available packages even as they develop packages for private use. This policy can be implemented with an override bazaar combining the local bazaar with one or more public bazaars.

Localized versions of packages Local groups may want to use something similar to a widely scoped bazaar, but override specific packages with localized versions. For example, they might prefer a different default language settings of the packages than is used on the widely scoped bazaar. Users want to use the localized version of a package whenever one is available, but the global version otherwise. This policy can be implemented by creating a bazaar server holding the localized packages, and then having users operate in an override bazaar combining this localized bazaar with the public one.

Stable versus unstable streams Projects frequently distinguish between stable and unstable streams of development. The stable streams include packages that are heavily tested and deemed to be reliable, while the unstable streams include packages that are more current and featureful but are not as reliable. A project can implement this policy by having separate bazaar servers for each development stream. Users point their managed directories to the bazaar server appropriate to their needs.

Freezing new stable distribution A common process for generating stable distributions of code is to take an unstable stream, start testing it, and disallow any patches except for bug fixes. After some point, the distribution is *frozen* and considered a stable release. Such processes can be implemented by manipulating the outstanding keys as time passes. The testing phase can be implemented by revoking all outstanding edit keys and switching to a moderation queue process. The final, complete freeze can be implemented by destroying the moderation queue and revoking the remaining edit keys. The frozen bazaar can then be duplicated, with one fork hosting a new development stream while the other becomes is designated a stable release.

6 Packages and Package Advertisements

6.1 Overview

A Scala Bazaars package is stored in a file with file-ending `.sbp` (Scala Bazaars Package). The file is in zip format. It should include all of the files that are to be extracted into a managed directory when the package is installed.

The `meta/` directory within a `.sbp` file does not include files to be extracted, but instead contains meta-information about the package itself. The only defined element of that directory is `meta/description`, which should include an XML document describing the package's contents. All other names within the `meta` subdirectory are reserved for future use.

6.2 Package Description Files

An example description file is as follows:

```
<package>
  <name>sbaz</name>
  <version>1.10</version>
  <depends> </depends>
  <description>The command-line interface to Scala Bazaars.  Scala
Bazaars let you share Scala packages and other goodies with other
Scala users.</description>
</package>
```

The elements of this description are hopefully self-explanatory. This package is named "sbaz", its version is 1.10, it has no dependencies, and it has the given human-readable description.

A package name should only include alphanumeric characters and hyphens.

6.3 Package Advertisements

A package advertisement is simply a package description plus a URL where the package file can be downloaded from. An example of the XML format for a package advertisement is:

```
<availablePackage>
  <package>
    <name>sbaz</name>
    <version>1.10</version>
    <depends> </depends>
    <description>The command-line interface to Scala Bazaars.  Scala
Bazaars let you share Scala packages and other goodies with other
Scala users.</description>
  </package>
  <link>http://lamp.epfl.ch/~spoon/scbaztmp/sbaz-1.10.sbp</link>
</availablePackage>
```

7 Versions and Dependencies

Every package has a version. A version is a string that may contain ASCII digits, alphabetic characters, and the following symbolic characters:

```
.-+/,@
```

Versions are totally ordered by the following algorithm. To compare two versions, first break them into a number of subsequences, each maximally long, where each subsequence only contains decimal digits, only contains alphabetic characters, or only contains symbolic characters. Compare the two sequences lexicographically. Two numeric subsequences are compared numerically, two alphabetic subsequences are compared in ASCII order, and two symbolic subsequences are compared in ASCII order as well. For two subsequences with different kinds of characters, the order is:

alphabetic, then numeric, then symbolic. For some examples, the following versions are sorted in order:

1. (empty string)
2. abc
3. abcd
4. abd
5. 1
6. 1.1
7. 1.1a
8. 1.1a2
9. 1.1a100
10. 1.1.5
11. 1.2
12. 1.2.
13. 2
14. 12

A package's dependencies are a list of other packages' names. For the dependencies to be satisfied, packages with the specified names must be installed. Here is an example package description for a package that depends on packages "scala2-library" and "sbaz":

```
<availablePackage>
  <package>
    <name>base</name>
    <version>1.7</version>
    <depends>
       <name>scala2-library</name>
       <name>sbaz</name>
    </depends>
    <description>This package depends on the basic packages that all
managed directories must include.  Each of these packages is either
essential or is very commonly used.  </description>
  </package>
  <link>http://lamp.epfl.ch/~spoon/scbaztmp/base-1.7.sbp</link>
</availablePackage>
```

A richer set of dependencies is planned for the future, including provides, suggests, alternative dependencies, and minimum version numbers.

8 Command-Line Interface

8.1 Overview

The command-line interface to Scala Bazaars is the `sbaz` tool. It is run as follows:

sbaz [-n] [-d *dir*] *command arguments...*

It always operates on one managed directory. That directory can be specified explicitly with the -d option. If it is not specified, and the `sbaz` binary is located within a managed directory, then `sbaz` operates on the managed directory the binary is in. If neither of these cases hold, then `sbaz` will as a last resort operate on the current directory. In all cases, the tool checks whether the specified directory appears in fact to be a managed directory; if it does not, then it aborts.

If -n is specified, then the tool does not perform any work. Instead, it only prints out what it would have done without the -n option.

If no *command* is specified, then the tool prints out a help message. Otherwise, it runs the specified command with the specified arguments.

8.2 Command Reference

This section will eventually include the entire command reference of the command-line tool. It does not yet. Refer to "`sbaz help`" to see the tool's built-in documentation.

9 Suggested Directory Layout

Each sbaz repository has its own standards for the directory layout within a managed directory. This section documents the emerging layout used in the `scala-dev`Scala bazaar. It is the standard for that repository, and it might serve as a guideline for other repositories.

- `lib` — Any jar file(s) associated with the package, especially those that are meant as libraries to be usable by other programs in the bazaar. Jars placed in this directory are particularly easy to access, because both the generic `scala` script and most of the tool-running scripts in `bin` will automatically load classes from any jars in `lib`. Normally, jar filenames in this directory do not include any version number. A typical filename is `sbaz.jar`.
- `src` — Source code for the package. This source code should be presented in a way that IDE's can find the code easily. One good option is to include a jar file of each package's source code in a file ending with `-src.jar`. For example, class `sbaz.clui.CommandLine` is found within file `src/sbaz-src.jar`.
- `bin` — Command-line runnable scripts. These are most easily created via the Scala ant tasks. As a special case, the `sbaz` tool will make files within `bin` be executable on platforms where that makes sense.
- `config` — Configuration files. Packages should not include any files in this directory! They should look in this directory for optional user configuration. If there is a single configuration file, it can be included directly in the `config` directory, e.g. with a name like `config/sbaz.properties`. If there is more than one configuration file for a package, then the files should be located in a subdirectory of `config` named after the package name. For example, the `sbaz` package could include its configuration files in a directory named `config/sbaz/`.
- `misc` — Arbitrary files not included in any of the above. All such files for a package should be included in a directory named after the package. For example, the `sbaz` package includes miscellaneous files in the directory `misc/sbaz/`.

- `doc` — Documentation files. All such files for a package should be included in a directory named after the package. For example, the `sbaz` package includes documentation files in the directory `doc/sbaz`.
- `man` — Unix man pages, with the usual man1, etc., subdirectories.

10 The scala.home property

By convention, the `scala.home` system property is used to point at the root of the current managed directory. Command-line scripts installed in `bin`directories should normally set this variable. The conventional choice is to set `scala.home` to the `SCALA_HOME` environment variable if it set, or otherwise to the parent directory of the `bin`directory the script is located in.

Programs intended to run within a sbaz directory can use this property to locate any files they may need. For example, `sbaz` itself uses the following code to find a user-specified settings file:

```
val home = System.getProperty("scala.home", ".")
val propFile = new File(new File(new File(home),
                        "config"), "sbaz.properties")
```

11 Managed Directory Layout

Most of a managed directory is populated by the contents of packages. The single directory `meta`is reserved for the system to track information about the managed directory. The `meta` directory has the following contents:

- `universe` — a file holding the XML description of the bazaar.
- `available` — a list of package advertisements available within the bazaar. The file is an XML document whose top level node is `<availableList>` and whose subnodes are package advertisements.
- `cached` — a subdirectory holding a cache of package files that the tool has downloaded.
- `installed` — information about installed files. The format is described below.
- `keyring.`*name*— the keyring holding known keys for the universe named *name*. There is one keyring file for each universe with locally known keys.

The `installed` file holds information about the packages that are currently installed in the managed directory. Its format is mostly straightforward:

```
<installedlist>
  <installedpackage>
    <package>
      <name>base</name>
      <version>1.7</version>
      <depends>
        <name>sbaz</name>
        <name>scala2-library</name>
      </depends>
      <description>A package that depends on
the basic, necessary packages</description>
```

```
    </package>
    <files>
      <filename isAbsolute="true" isFile="true">
        <pathcomp>lib</pathcomp>
      </filename>
    </files>
  </installedpackage>

  <installedpackage>
    <package>
      <name>sbaz</name>
      <version>1.10</version>
      <depends></depends>
      <description>The command-line interface to Scala Bazaars.</description>
    </package>
    <files>
      <filename isAbsolute="true" isFile="true">
        <pathcomp>bin</pathcomp>
        <pathcomp>sbaz</pathcomp>
      </filename>

      <filename isAbsolute="true" isFile="true">
        <pathcomp>bin</pathcomp>
        <pathcomp>sbaz.bat</pathcomp>
      </filename>

      <filename isAbsolute="true" isFile="true">
        <pathcomp>misc</pathcomp>
        <pathcomp>sbaz</pathcomp>
        <pathcomp>scala2-library.jar</pathcomp>
      </filename>

      <filename isAbsolute="true" isFile="true">
        <pathcomp>misc</pathcomp>
        <pathcomp>sbaz</pathcomp>
        <pathcomp>smartrun.mswin.template</pathcomp>
      </filename>

      <filename isAbsolute="true" isFile="true">
        <pathcomp>misc</pathcomp>
        <pathcomp>sbaz</pathcomp>
        <pathcomp>smartrun.unix.template</pathcomp>
      </filename>

      <filename isAbsolute="true" isFile="true">
        <pathcomp>lib</pathcomp>
        <pathcomp>sbaz.jar</pathcomp>
      </filename></files>
    </installedpackage>
  <installedpackage>
</installedist>
```

Notice that an installed package is the combination of a package plus a list of files. Each file in the list of files is represented with a `filename` whose attributes determine whether it is an absolute file (never) and whether it is a file (versus a directory). The contents of the `filename` are the sequence of path components of the file, relative to the managed directory's root.

12 Common Problems

12.1 Firewalls and HTTP Proxies

Scala Bazaars uses HTTP to communicate with universe servers. If your network blocks HTTP access, then, you need to configure sbaz to use an HTTP proxy. To do this, create a file named `config/sbaz.properties` in your managed directory and give it the appropriate proxy settings, something like:

```
http.proxySet=true
http.proxyHost=localhost
http.proxyPort=3128
```