```
**********************************************************
   12650 Tue Sep  9 16:17:33 2008
new/src/modules/server/catalog.py
3166 feed generation needs performance improvement
3306 feed returns invalid last-modified header
**********************************************************
```

```
#!/usr/bin/python2.4
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.

import subprocess
import threading
import signal
import os
import sys
import cherrypy

import pkg.catalog as catalog
import pkg.fmri as fmri
#endif /* ! codereview */
import pkg.indexer as indexer
import pkg.server.query_engine as query_e

from pkg.misc import SERVER_DEFAULT_MEM_USE_KB
from pkg.misc import emsg

class ServerCatalog(catalog.Catalog):
        """The catalog information which is only needed by the server."""

        def __init__(self, cat_root, authority = None, pkg_root = None,
            read_only = False, index_root = None, repo_root = None,
            rebuild = True):

                self.index_root = index_root
                self.repo_root = repo_root

                # The update_handle lock protects the update_handle variable.
                # This allows update_handle to be checked and acted on in a
                # consistent step, preventing the dropping of needed updates.
                # The check at the top of refresh index should always be done
                # prior to deciding to spin off a process for indexing as it
                # prevents more than one indexing process being run at the same
                # time.
                self.searchdb_update_handle_lock = threading.Lock()

                if self.index_root:
```

```
                        self.query_engine = \
                            query_e.ServerQueryEngine(self.index_root)

                if os.name == 'posix':
                        try:
                                signal.signal(signal.SIGCHLD,
                                    self.child_handler)
                        except ValueError:
                                emsg("Tried to create signal handler in "
                                    "a thread other than the main thread")

                self.searchdb_update_handle = None
                self._search_available = False
                self.deferred_searchdb_updates = []
                self.deferred_searchdb_updates_lock = threading.Lock()

                self.refresh_again = False

                catalog.Catalog.__init__(self, cat_root, authority, pkg_root,
                    read_only, rebuild)

                if not self._search_available:
                        self._check_search()

        def whence(self, cmd):
                if cmd[0] != '/':
                        tmp_cmd = cmd
                        cmd = None
                        path = os.environ['PATH'].split(':')
                        path.append(os.environ['PWD'])
                        for p in path:
                                if os.path.exists(os.path.join(p, tmp_cmd)):
                                        cmd = os.path.join(p, tmp_cmd)
                                        break
                        assert cmd
                return cmd

        def refresh_index(self):
                """ This function refreshes the search indexes if there any new
                packages. It starts a subprocess which results in a call to
                run_update_index (see below) which does the actual update.
                """

                self.searchdb_update_handle_lock.acquire()

                if self.searchdb_update_handle:
                        self.refresh_again = True
                        self.searchdb_update_handle_lock.release()
                        return

                try:
                        fmris_to_index = set(self.fmris())

                        indexer.Indexer.check_for_updates(self.index_root,
                            fmris_to_index)

                        if fmris_to_index:
                                if os.name == 'posix':
                                        cmd = self.whence(sys.argv[0])
                                        args = (cmd, "--refresh-index", "-d",
                                            self.repo_root)
                                        try:
                                                self.searchdb_update_handle = \
                                                    subprocess.Popen(args,
                                                        stderr = \
                                                        subprocess.STDOUT)
                                        except Exception, e:
                                                emsg("Starting the indexing "
                                                    "process failed")
                                                raise
                                else:
                                        self.run_update_index()
                        else:
                                # Since there is nothing to index, setup
                                # the index and declare search available.
                                # We only log this if this represents
                                # a change in status of the server.
                                ind = indexer.Indexer(self.index_root,
                                    SERVER_DEFAULT_MEM_USE_KB)
                                ind.setup()
                                if not self._search_available:
                                        cherrypy.log("Search Available",
                                            "INDEX")
                                self._search_available = True
                finally:
                        self.searchdb_update_handle_lock.release()

        def run_update_index(self):
                """ Determines which fmris need to be indexed and passes them
                to the indexer.

                Note: Only one instance of this method should be running.
                External locking is expected to ensure this behavior. Calling
                refresh index is the preferred method to use to reindex.
                """
                fmris_to_index = set(self.fmris())

                indexer.Indexer.check_for_updates(self.index_root,
                    fmris_to_index)

                if fmris_to_index:
                        self.__update_searchdb_unlocked(fmris_to_index)
                else:
                        ind = indexer.Indexer(self.index_root,
                            SERVER_DEFAULT_MEM_USE_KB)
                        ind.setup()

        def _check_search(self):
                ind = indexer.Indexer(self.index_root,
                    SERVER_DEFAULT_MEM_USE_KB)
                if ind.check_index_existence():
                        self._search_available = True
                        cherrypy.log("Search Available", "INDEX")

        def build_catalog(self):
                """ Creates an Indexer instance and after building the
                catalog, refreshes the index.
                """
                self._check_search()
                catalog.Catalog.build_catalog(self)
                # refresh_index doesn't use file modification times
                # to determine which packages need to be indexed, so use
                # it to reindex if it's needed.
                self.refresh_index()

        def child_handler(self, sig, frame):
                """ Handler method for the SIGCLD signal.  Checks to see if the
                search database update child has finished, and enables searching
                if it finished successfully, or logs an error if it didn't.
                """
                try:
                        signal.signal(signal.SIGCHLD, self.child_handler)
```

```
                except ValueError:
                        emsg("Tried to create signal handler in "
                            "a thread other than the main thread")
                # If there's no update_handle, then another subprocess was
                # spun off and that was what finished. If the poll() returns
                # None, then while the indexer was running, another process
                # that was spun off finished.
                rc = None
                if not self.searchdb_update_handle:
                        return
                rc = self.searchdb_update_handle.poll()
                if rc == None:
                        return

                if rc == 0:
                        self._search_available = True
                        cherrypy.log("Search indexes updated and available.",
                            "INDEX")
                        # Need to acquire this lock to prevent the possibility
                        # of a race condition with refresh_index where a needed
                        # refresh is dropped. It is possible that an extra
                        # refresh will be done with this code, but that refresh
                        # should be very quick to finish.
                        self.searchdb_update_handle_lock.acquire()
                        self.searchdb_update_handle = None
                        self.searchdb_update_handle_lock.release()

                        if self.refresh_again:
                                self.refresh_again = False
                                self.refresh_index()
                elif rc > 0:
                        # XXX This should be logged instead
                        # If the refresh of the index failed, defensively
                        # declare that search is unavailable.
                        self._search_available = False
                        emsg(_("ERROR building search database, rc: %s"))
                        emsg(_(self.searchdb_update_handle.stderr.read()))

        def __update_searchdb_unlocked(self, fmri_list):
                """ Takes a fmri_list and calls the indexer with a list of fmri
                and manifest file path pairs. It assumes that all needed
                locking has already occurred.
                """
                assert self.index_root
                fmri_manifest_list = []

                # Rather than storing those, simply pass along the
                # file and have the indexer take care of opening and
                # reading the manifest file. Since the indexer
                # processes and discards the manifest structure (and its
                # search dictionary for that matter) this
                # is much more memory efficient.

                for f in fmri_list:
                        mfst_path = os.path.join(self.pkg_root,
                                                 f.get_dir_path())
                        fmri_manifest_list.append((f, mfst_path))

                if fmri_manifest_list:
                        index_inst = indexer.Indexer(self.index_root,
                            SERVER_DEFAULT_MEM_USE_KB)
                        index_inst.server_update_index(fmri_manifest_list)

        def search(self, token):
                """Search through the search database for 'token'.  Return a
                list of token type / fmri pairs."""
                assert self.index_root
                if not self.query_engine:
                        self.query_engine = \
                            query_e.ServerQueryEngine(self.index_root)
                query = query_e.Query(token, case_sensitive=False)
                return self.query_engine.search(query)

        @staticmethod
        def read_catalog(catalog, dir, auth=None):
                """Read the catalog file in "dir" and combine it with the
                existing data in "catalog"."""

                catf = file(os.path.join(dir, "catalog"))
                for line in catf:
                        if not line.startswith("V pkg") and \
                            not line.startswith("C pkg"):
                                continue

                        f = fmri.PkgFmri(line[7:])
                        ServerCatalog.cache_fmri(catalog, f, auth)

                catf.close()

#endif /* ! codereview */
```

```
**********************************************************
   16866 Tue Sep  9 16:17:34 2008
new/src/modules/server/feed.py
3166 feed generation needs performance improvement
3306 feed returns invalid last-modified header
**********************************************************
#!/usr/bin/python2.4
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.

"""feed - routines for generating RFC 4287 Atom feeds for packaging server

   At present, the pkg.server.feed module provides a set of routines that, from
   a catalog, allow the construction of a feed representing the activity within
   a given time period."""

import cherrypy
from cherrypy.lib.static import serve_file
import cStringIO
import datetime
import httplib
import os
import rfc822
import sys
#endif /* ! codereview */
import time
import urllib
import xml.dom.minidom as xmini

from pkg.misc import get_rel_path, get_res_path
import pkg.server.catalog as catalog
import pkg.catalog as catalog
import pkg.fmri as fmri
import pkg.Uuid25 as uuid

MIME_TYPE = 'application/atom+xml'
CACHE_FILENAME = "feed.xml"
RFC3339_FMT = "%Y-%m-%dT%H:%M:%SZ"

def dt_to_rfc3339_str(ts):
        """Returns a string representing a datetime object formatted according
        to RFC 3339.
        """
        return ts.strftime(RFC3339_FMT)

def rfc3339_str_to_ts(ts_str):
```

```
        """Returns a timestamp representing 'ts_str', which should be in the
        format specified by RFC 3339.
        """
        return time.mktime(time.strptime(ts_str, RFC3339_FMT))

def rfc3339_str_to_dt(ts_str):
        """Returns a datetime object representing 'ts_str', which should be in
        the format specified by RFC 3339.
        """
        return datetime.datetime(*time.strptime(ts_str, RFC3339_FMT)[0:6])

def ults_to_ts(ts_str):
        """Returns a timestamp representing 'ts_str', which should be in
        updatelog format.
        """
        # Python doesn't support fractional seconds for strptime.
        ts_str = ts_str.split('.')[0]
        # Currently, updatelog entries are in local time, not UTC.
        return time.mktime(time.strptime(ts_str, "%Y-%m-%dT%H:%M:%S"))

def ults_to_rfc3339_str(ts_str):
        """Returns a timestamp representing 'ts_str', which should be in
        updatelog format.
        """
        ltime = ults_to_ts(ts_str)
        # Currently, updatelog entries are in local time, not UTC.
        return dt_to_rfc3339_str(datetime.datetime(
            *time.gmtime(ltime)[0:6]))

def fmri_to_taguri(rcfg, f):
        """Generates a 'tag' uri compliant with RFC 4151.  Visit
        http://www.taguri.org/ for more information.
        """
        return "tag:%s,%s:%s" % (rcfg.get_attribute("feed",
            "authority"), f.get_timestamp().strftime("%Y-%m-%d"),
            urllib.unquote(f.get_url_path()))

def init(scfg, rcfg):
        """This function performs general initialization work that is needed
        for feeds to work correctly.
        """

        if not scfg.is_read_only():
                # RSS/Atom feeds require a unique identifier, so
                # generate one if isn't defined already.  This
                # needs to be a persistent value, so we only
                # generate this if we can save the configuration.
                fid = rcfg.get_attribute("feed", "id")
                if not fid:
                        # Create a random UUID (type 4).
                        rcfg._set_attribute("feed", "id", uuid.uuid4())

                # Ensure any configuration changes are reflected in the feed.
                __clear_cache(scfg)

def set_title(request, rcfg, doc, feed, update_ts):
        """This function attaches the necessary RSS/Atom feed elements needed
        to provide title, author and contact information to the provided
        xmini document object using the provided feed object and update
        time.
        """

        t = doc.createElement("title")
        ti = xmini.Text()
        ti.replaceWholeText(rcfg.get_attribute("feed", "name"))
        t.appendChild(ti)
        feed.appendChild(t)

        l = doc.createElement("link")
        l.setAttribute("href", cherrypy.url())
        l.setAttribute("rel", "self")
        feed.appendChild(l)

        # Atom requires each feed to have a permanent, universally unique
        # identifier.
        i = doc.createElement("id")
        it = xmini.Text()
        it.replaceWholeText("urn:uuid:%s" % rcfg.get_attribute("feed", "id"))
        i.appendChild(it)
        feed.appendChild(i)

        # Indicate when the feed was last updated.
        u = doc.createElement("updated")
        ut = xmini.Text()
        ut.replaceWholeText(dt_to_rfc3339_str(update_ts))
        u.appendChild(ut)
        feed.appendChild(u)

        # Add our icon.
        i = doc.createElement("icon")
        it = xmini.Text()
        it.replaceWholeText(get_res_path(request, rcfg.get_attribute(
            "feed", "icon")))
        i.appendChild(it)
        feed.appendChild(i)

        # Add our logo.
        l = doc.createElement("logo")
        lt = xmini.Text()
        lt.replaceWholeText(get_res_path(request, rcfg.get_attribute(
            "feed", "logo")))
        l.appendChild(lt)
        feed.appendChild(l)

        maintainer = rcfg.get_attribute("repository", "maintainer")
        # The author information isn’t required, but can be useful.
        if maintainer:
                name, email = rfc822.AddressList(maintainer).addresslist[0]

                if email and not name:
                        # If we got an email address, but no name, then
                        # the name was likely parsed as a local address. In
                        # that case, assume the whole string is the name.
                        name = maintainer
                        email = None

                a = doc.createElement("author")

                # First we have to add a name element. This is required if an
                # author element exists.
                n = doc.createElement("name")
                nt = xmini.Text()
                nt.replaceWholeText(name)
                n.appendChild(nt)
                a.appendChild(n)

                if email:
                        # If we were able to extract an email address from the
                        # maintainer information, add the optional email
                        # element to provide a point of communication.
                        e = doc.createElement("email")
                        et = xmini.Text()
```

```
                        et.replaceWholeText(email)
                        e.appendChild(et)
                        a.appendChild(e)

                # Done with the author.
                feed.appendChild(a)

operations = {
        "+": ["Added", "%s was added to the repository."],
        "-": ["Removed", "%s was removed from the repository."],
        "U": ["Updated", "%s, an update to an existing package, was added to "
            "the repository."]
}

def add_transaction(request, scfg, rcfg, doc, feed, txn, fmris):
def add_transaction(request, scfg, rcfg, doc, feed, txn):
        """Each transaction is an entry.  We have non-trivial content, so we
        can omit summary elements.
        """

        e = doc.createElement("entry")

        tag, fmri_str = txn["catalog"].split()
        f = fmri.PkgFmri(fmri_str)

        # Generate a ’tag’ uri, to uniquely identify the entry, using the fmri.
        i = xmini.Text()
        i.replaceWholeText(fmri_to_taguri(rcfg, f))
        eid = doc.createElement("id")
        eid.appendChild(i)
        e.appendChild(eid)

        # Attempt to determine the operation that was performed and generate
        # the entry title and content.
        if txn["operation"] in operations:
                op_title, op_content = operations[txn["operation"]]
        else:
                # XXX Better way to reflect an error?  (Aborting will make a
                # non-well-formed document.)
                op_title = "Unknown Operation"
                op_content = "%s was changed in the repository."

        if txn["operation"] == "+":
                c = scfg.updatelog.catalog
                # Get all FMRIs matching the current FMRI’s package name.
                matches = fmris[f.pkg_name]
                if len(matches["versions"]) > 1:
                        # Get the oldest fmri.
                        of = matches[str(matches["versions"][0])][0]
                matches = catalog.extract_matching_fmris(c.fmris(),
                    f.get_name(), matcher=fmri.exact_name_match)

                if len(matches) > 1:
                        # Get the oldest fmri (it’s the last entry).
                        of = matches[-1]

                        # If the current fmri isn’t the oldest one, then this
                        # is an update to the package.
                        if f != of:
                                # If there is more than one matching FMRI, and
                                # it isn’t the same version as the oldest one,
                                # we can assume that this is an update to an
                                # existing package.
                                op_title, op_content = operations["U"]

        # Now add a title for our entry.
        etitle = doc.createElement("title")
        ti = xmini.Text()
        ti.replaceWholeText(" ".join([op_title, fmri_str]))
        etitle.appendChild(ti)
        e.appendChild(etitle)

        # Indicate when the entry was last updated (in this case, when the
        # package was added).
        eu = doc.createElement("updated")
        ut = xmini.Text()
        ut.replaceWholeText(ults_to_rfc3339_str(txn["timestamp"]))
        eu.appendChild(ut)
        e.appendChild(eu)

        # Link to the info output for the given package FMRI.
        e_uri = get_rel_path(request, 'info/0/%s' % f.get_url_path())

        l = doc.createElement("link")
        l.setAttribute("rel", "alternate")
        l.setAttribute("href", e_uri)
        e.appendChild(l)

        # Using the description for the operation performed, add the FMRI and
        # tag information.
        content_text = op_content % fmri_str
        if tag == "C":
                content_text += "  This version is tagged as critical."

        co = xmini.Text()
        co.replaceWholeText(content_text)
        ec = doc.createElement("content")
        ec.appendChild(co)
        e.appendChild(ec)

        feed.appendChild(e)

def update(request, scfg, rcfg, t, cf):
        """Generate new Atom document for current updates.  The cached feed
        file is written to scfg.repo_root/CACHE_FILENAME.
        """

        # Our configuration is stored in hours, convert it to seconds.
        window_seconds = rcfg.get_attribute("feed", "window") * 60 * 60
        feed_ts = datetime.datetime.fromtimestamp(t - window_seconds)

        d = xmini.Document()

        feed = d.createElementNS("http://www.w3.org/2005/Atom", "feed")
        feed.setAttribute("xmlns", "http://www.w3.org/2005/Atom")

        set_title(request, rcfg, d, feed, scfg.updatelog.last_update)

        d.appendChild(feed)

        # The feed should be presented in reverse chronological order.
        def compare_ul_entries(a, b):
                return cmp(ults_to_ts(a["timestamp"]),
                    ults_to_ts(b["timestamp"]))

        # Get the entire catalog in the format returned by catalog.cache_fmri,
        # so that we don't have to keep looking for possible matches.
        fmris = {}
        catalog.ServerCatalog.read_catalog(fmris,
            scfg.updatelog.catalog.catalog_root)

#endif /* ! codereview */
```

```
        for txn in sorted(scfg.updatelog.gen_updates_as_dictionaries(feed_ts),
            cmp=compare_ul_entries, reverse=True):
                add_transaction(request, scfg, rcfg, d, feed, txn, fmris)
                add_transaction(request, scfg, rcfg, d, feed, txn)

        d.writexml(cf)

def __get_cache_pathname(scfg):
        return os.path.join(scfg.repo_root, CACHE_FILENAME)

def __clear_cache(scfg):
        if scfg.is_read_only():
                # Ignore the request due to server configuration.
                return

        pathname = __get_cache_pathname(scfg)
        try:
                if os.path.exists(pathname):
                        os.remove(pathname)
        except IOError:
                raise cherrypy.HTTPError(
                    httplib.INTERNAL_SERVER_ERROR,
                    "Unable to clear feed cache.")

def __cache_needs_update(scfg):
        """Checks to see if the feed cache file exists and if it is still
        valid.  Returns False, None if the cache is valid or True, last
        where last is a timestamp representing when the cache was
        generated.
        """
        cfpath = __get_cache_pathname(scfg)
        last = None
        need_update = True
        if os.path.isfile(cfpath):
                # Attempt to parse the cached copy.  If we can't, for any
                # reason, assume we need to remove it and start over.
                try:
                        d = xmini.parse(cfpath)
                except Exception:
                        d = None
                        __clear_cache(scfg)

                # Get the feed element and attempt to get the time we last
                # generated the feed to determine whether we need to regenerate
                # it.  If for some reason we can't get that information, assume
                # the cache is invalid, clear it, and force regeneration.
                fe = None
                if d:
                        fe = d.childNodes[0]

                if fe:
                        utn = None
                        for cnode in fe.childNodes:
                                if cnode.nodeName == "updated":
                                        utn = cnode.childNodes[0]
                                        break

                        if utn:
                                last_ts = rfc3339_str_to_dt(utn.nodeValue)

                                # Since our feed cache and updatelog might have
                                # been created within the same second, we need
                                # to ignore small variances when determining
                                # whether to update the feed cache.
                                update_ts = scfg.updatelog.last_update.replace(
                                    microsecond=0)
                                if last_ts >= update_ts:
                                        need_update = False
                                else:
                                        last = rfc3339_str_to_ts(utn.nodeValue)
                        else:
                                __clear_cache(scfg)
                else:
                        __clear_cache(scfg)

        return need_update, last

def handle(scfg, rcfg, request, response):
        """If there have been package updates since we last generated the feed,
        update the feed and send it to the client.  Otherwise, send them the
        cached copy if it is available.
        """

        cfpath = __get_cache_pathname(scfg)

        # First check to see if we already have a valid cache of the feed.
        need_update, last = __cache_needs_update(scfg)

        if need_update:
                # Update always looks at feed.window seconds before the last
                # update until "now."  If last is none, we want it to use "now"
                # as its starting point.
                if last is None:
                        last = time.time()

                if scfg.is_read_only():
                        # If the server is operating in readonly mode, the
                        # feed will have to be generated every time.
                        cf = cStringIO.StringIO()
                        update(request, scfg, rcfg, last, cf)
                        cf.seek(0)
                        buf = cf.read()
                        cf.close()

                        # Now that the feed has been generated, set the headers
                        # correctly and return it.
                        response.headers['Content-type'] = MIME_TYPE

                        # Return the current time and date in GMT.
                        response.headers['Last-Modified'] = rfc822.formatdate()

                        response.headers['Last-Modified'] = \
                            datetime.datetime.now().isoformat()
                        response.headers['Content-length'] = len(buf)
                        return buf
                else:
                        # If the server isn't operating in readonly mode, the
                        # feed can be generated and cached in inst_dir.
                        cf = file(cfpath, "w")
                        update(request, scfg, rcfg, last, cf)
                        cf.close()

        return serve_file(cfpath, MIME_TYPE)
```