Heritrix developer documentation

Internet ArchiveEdited by John Erik Halse
Gordon Mohr
Kristinn Sigur#sson
Michael Stack
Paul Jack

Table of Contents

1. Introduction 2
2. Obtaining and building Heritrix 2
2.1. Obtaining Heritrix 2
2.2. Building Heritrix 2
2.3. Running Heritrix 3
2.4. Eclipse 3
2.5. Integration self test 3
3. Coding conventions 4
3.1. Tightenings on the SUN conventions 4
3.2. Long versus int 5
3.3. Unit tests code in same package 5
3.4. Log Message Format 5
4. Overview of the crawler 5
4.1. The CrawlController 6
4.2. The Frontier 6
4.3. ToeThreads 6
4.4. Processors 6
5. Settings 8
5.1. Settings hierarchy 8
5.2. ComplexType hierarchy 9
6. Common needs for all configurable modules 10
6.1. Definition of a module 10
6.2. Accessing attributes 12
6.3. Putting together a simple module 12
7. Some notes on the URI classes 14
7.1. Supported Schemes (UnsupportedUriSchemeException) 14
7.2. The CrawlURI's Attribute list 14
7.3. The recorder streams 15
8. Writing a Frontier 15
9. Writing a Filter 21
10. Writing a Scope 21
11. Writing a Processor 22
11.1. Accessing and updating the CrawlURI 23
11.2. The HttpRecorder 25
11.3. An example processor 26
11.4. Things to keep in mind when writing a processor 27
12. Writing a Statistics Tracker 28
12.1. AbstractTracker 28
12.2. Provided StatisticsTracker 28
13. Internet Archive ARC files 29
13.1. ARC File Naming 29

13.2. Reading arc files 30
13.3. Writing arc files 30
13.4. Searching ARCS 30
A. Future changes in the API 30
1. The org.archive.util.HTTPRecorder class 30
2. The Frontiers handling of dispositions 31
B. Version and Release Numbering 31
C. Making a Heritrix Release 32
D. Settings XML Schema 33
E. Profiling Heritrix 33
Bibliography 33

1. Introduction

This manual is intended to be a starting point for users and contributors who wants to learn about the internals of the Heritrix web crawler and possibly write additions or contribute to the core of the software. The javadoc API documentation [http://crawler.archive.org/apidocs/index.html] is supposed to be the main developer documentation, but this document should help you get started and guide you to the interesting parts of the Javadoc documentation.

2. Obtaining and building Heritrix

2.1. Obtaining Heritrix

Heritrix can be obtained as packaged binary or source downloaded from the crawler sourceforge home page [http://sourceforge.net/projects/archive-crawler], or via checkout from archive-crawler.svn.sourceforge.net. See the crawler sourceforge svn page [http://sourceforge.net/svn/?group_id=73833] for how to fetch from subversion. The `Module Name` name to use checking out heritrix is `ArchiveOpenCrawler`, the name Heritrix had before it was called Heritrix.

Note

Note, anonymous access does not give you the current HEAD but a snapshot that can some times be up to 24 hours behind HEAD.

The packaged binary is named heritrix-?.?.?.tar.gz (or heritrix-?.?.?.zip) and the packaged source is named heritrix-?.?.?-src.tar.gz (or heritrix-?.?.?-src.zip) where ?.?.? is the heritrix release version.

2.2. Building Heritrix

You can build Heritrix from source using Maven. Heritrix build has been tested against maven-1.0.2. Do not use Maven 2.x to build Heritrix. See maven.apache.org [http://maven.apache.org] for how to obtain the binary and setup of your maven environment.

In addition to the base maven build, if you want to generate the docbook user and developer manuals, you will need to add the maven sdocbook plugin which can be found at this page [http://maven-plugins.sourceforge.net/maven-sdocbook-plugin/downloads.html] (If the sdocbook plugin is not present, the build skips the docbook manual generation). Be careful. Do not confuse the 'sdocbook' plugin with the similarly named 'docbook' plugin. This latter converts docbook to xdocs where what's wanted is the former, convert docbook xml to html. This 'sdocbook' plugin is used to generate the user and developer documentation.

Download the plugin jar -- currently, as of this writing, its maven-sdocbook-plugin-1.4.1.jar -- and put it into your maven repository plugins directory, usually at `${MAVEN_HOME}/plugins/` (in earlier versions of maven, pre 1.0.2, plugins are at `${HOME}/.maven/plugins/`).

The sdocbook plugin has a dependency on the jimi jar from sun [http://java.sun.com/products/jimi/] which you will have to manually pull down and place into your maven respository (Its got a sun license you must accept so maven cannot autodownload). Download the jimi package and unzip it. Rename the file named `JimiProClasses.zip` as `jimi-1.0.jar` and put it into your maven jars repository (Usually `.maven/repository/jimi/jars`. You may have to create the later directories manually). Maven will be looking for a jar named jimi-1.0.jar. Thats why you have to rename the jimi class zip (jars are effectively zips).

Note

It may be necessary to alter the `sdocbook-plugin` default configuration. By default, sdocbook will download the latest version of `docbook-xsl`. However, sdocbook hardcodes a specific version number for `docbook-xsl` in its `plugin.properties` file. If you get an error like "Error while expanding ~/.maven/repository/docbook/zips/docbook-xsl-1.66.1.zip", then you will have to manually edit sdocbook's properties. First determine the version of `docbook-xsl` that you have -- it's in `~/.maven/repository/docbook/zips`. Once you have the version number, edit `~/.maven/cache/maven-sdocbook-plugin-1.4/plugin-properties` and change the `maven.sdocbook.stylesheets.version` property to the version that was actually downloaded.

To build a source checkout with Maven:

```
% cd CHECKOUT_DIR
% $MAVEN_HOME/bin/maven dist
```

In the target/distribution subdir, you will find packaged source and binary builds. Run $MAVEN_HOME/bin/maven -g for other Maven possibilities.

2.3. Running Heritrix

See the User Manual [Heritrix User Guide] for how to run the built Heritrix.

2.4. Eclipse

The development team uses Eclipse as the development environment. This is of course optional, but for those who want to use Eclipse, you can, at the head of the source tree, find Eclipse *.project* and *.classpath* configuration files that should make integrating the source checkout into your Eclipse development environment straight-forward.

When running direct from checkout directories, rather than a Maven build, be sure to use a JDK installation (so that JSP pages can compile). You will probably also want to set the 'heritrix.development' property (with the "-Dheritrix.development" VM command-line option) to indicate certain files are in their development, rather than deployment, locations.

2.5. Integration self test

Run the integration self test on the command line by doing the following:

```
% $HERITRIX_HOME/bin/heritrix --selftest
```

This will set the crawler going against itself, in particular, the selftest webapp. When done, it runs an analysis of the produced arc files and logs and dumps a ruling into `heritrix_out.log`. See the org.archive.crawler.selftest [http://crawler.archive.org/apidocs/org/archive/crawler/selftest/package-summary.html] package for more on how the selftest works.

3. Coding conventions

Heritrix baselines on SUN's Code Conventions for the Java Programming Language [Sun Code Conventions]. It'd be hard not to they say so little. They at least say maximum line length of 80 characters [http://java.sun.com/docs/codeconv/html/CodeConventions.doc3.html#313].

We also will favor much of what is written in the document, Java Programming Style Guidelines [Java Programming Style Guidelines].

3.1. Tightenings on the SUN conventions

Below are tightenings on the SUN conventions used in Heritrix.

3.1.1. No Tabs

No tabs in source code. Set your editor to indent with spaces.

3.1.2. Indent Width

Indents are 4 characters wide.

3.1.3. Function/Block Bracket Open on Same Line

Preference is to have the bracket that opens functions and blocks on same line as function declaration or block test rather than on a new line of its own. For example:

```
if (true) {
    return true;
}
```

and

```
public void main (String [] args) {
    System.println("Hello world");
}
```

3.1.4. File comment

Here is the eclipse template we use for the file header comment:

```
/* ${type_name}
 *
 * $$Id: developer_manual.xml 5022 2007-03-23 16:06:32Z stack-sf $$
 *
 * Created on ${date}
 *
 * Copyright (C) ${year} Internet Archive.
 *
 * This file is part of the Heritrix web crawler (crawler.archive.org).
 *
 * Heritrix is free software; you can redistribute it and/or modify
 * it under the terms of the GNU Lesser Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * any later version.
 *
 * Heritrix is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser Public License for more details.
 *
 * You should have received a copy of the GNU Lesser Public License
 * along with Heritrix; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
${package_declaration}
```

3.2. Long versus int

We will be performing multi-billion resource crawls -- which may have to pass up billions of pages that don't make the time/priority cut. Our access tools will range over tens if not hundreds of billions of resources. We may often archive resources larger than 2GB. Keep these in mind when choosing between 'int' (max value: around 2 billion) and 'long' (max value: around 9 quintillion) in your code.

3.3. Unit tests code in same package

"[A] popular convention is to place all test classes in a parallel directory structure. This allows you to use the same Java package names for your tests, while keeping the source files separate. To be honest, we do not like this approach because you must look in two different directories to find files." from *Section 4.11.3, Java Extreme Programming Cookbook, By Eric M. Burke, Brian M. Coyner*. We agree with the above so we put Unit Test classes beside the classes they are testing in the source tree giving them the name of the Class they are testing with a Test suffix.

Another advantage is that test classes of the same package can get at testee's default access methods and members, something a test in another package would not be able to do.

3.4. Log Message Format

A suggested format for source code log messages can be found at the subversion site, Log Messages [http://subversion.tigris.org/hacking.html#log-messages].

4. Overview of the crawler

The Heritrix Web Crawler is designed to be modular. Which modules to use can be set at runtime from the user interface. Our hope is that if you want the crawler to behave different from the default, it should only be a matter of writing a new module as a replacement or in addition to the modules shipped with the crawler.

The rest of this document assumes you have a basic understanding of how to run a crawl (see: [Heritrix User Guide]). Since the crawler is written in the Java programming language, you also need a fairly good understanding of Java.

The crawler consists of *core classes* and *pluggable modules*. The core classes can be configured, but not replaced. The pluggable classes can be substituted by altering the configuration of the crawler. A set of basic pluggable classes are shipped with the crawler, but if you have needs not met by these classes you could write your own.

Figure 1. Crawler overview

4.1. The CrawlController

The CrawlController collects all the classes which cooperate to perform a crawl, provides a high-level interface to the running crawl, and executes the "master thread" which doles out URIs from the Frontier to the ToeThreads. As the "global context" for a crawl, subcomponents will usually reach each other through the CrawlController.

4.2. The Frontier

The Frontier is responsible for handing out the next URI to be crawled. It is responsible for maintaining politeness, that is making sure that no web server is crawled too heavily. After a URI is crawled, it is handed back to the Frontier along with any newly discovered URIs that the Frontier should schedule for crawling.

It is the Frontier which keeps the state of the crawl. This includes, but is not limited to:

- What URIs have been discovered
- What URIs are being processed (fetched)
- What URIs have been processed

The Frontier implements the Frontier interface and can be replaced by any Frontier that implements this interface. It should be noted though that writing a Frontier is not a trivial task.

The Frontier relies on the behavior of at least the following external processors: PreconditionEnforcer, LinksScoper and the FrontierScheduler (See below for more each of these Processors). The PreconditionEnforcer makes sure dns and robots are checked ahead of any fetching. LinksScoper tests if we are interested in a particular URL -- whether the URL is 'within the crawl scope' and if so, what our level of interest in the URL is, the priority with which it should be fetched. The FrontierScheduler adds ('schedules') URLs to the Frontier for crawling.

4.3. ToeThreads

The Heritrix web crawler is multi threaded. Every URI is handled by its own thread called a ToeThread. A ToeThread asks the Frontier for a new URI, sends it through all the processors and then asks for a new URI.

4.4. Processors

Processors are grouped into processor chains (Figure 2, “Processor chains”). Each chain does some processing on a URI. When a Processor is finished with a URI the ToeThread sends the URI to the next Processor until the URI has been processed by all the Processors. A processor has the option of telling the URI to skip to a particular chain. Also if a processor throws a fatal error, the processing skips to the Post-processing chain.

Figure 2. Processor chains

The task performed by the different processing chains are as follows:

4.4.1. Pre-fetch processing chain

The first chain is responsible for investigating if the URI could be crawled at this point. That includes checking if all preconditions are met (DNS-lookup, fetching robots.txt, authentication). It is also possible to completely block the crawling of URIs that have not passed through the scope check.

In the `Pre-fetch processing` chain the following processors should be included (or replacement modules that perform similar operations):

- **Preselector**

  Last check if the URI should indeed be crawled. Can for example recheck scope. Useful if scope rules have been changed after the crawl starts. The scope is usually checked by the LinksScoper, before new URIs are added to the Frontier to be crawled. If the user changes the scope limits, it will not affect already queued URIs. By rechecking the scope at this point, you make sure that only URIs that are within current scope are being crawled.

- **PreconditionEnforcer**

  Ensures that all preconditions for crawling a URI have been met. These currently include verifying that DNS and robots.txt information has been fetched for the URI.

4.4.2. Fetch processing chain

The processors in this chain are responsible for getting the data from the remote server. There should be one processor for each protocol that Heritrix supports: e.g. FetchHTTP.

4.4.3. Extractor processing chain

At this point the content of the document referenced by the URI is available and several processors will in turn try to get new links from it.

4.4.4. Write/index processing chain

This chain is responsible for writing the data to archive files. Heritrix comes with an ARCWriterProcessor which writes to the ARC format. New processors could be written to support other formats and even create indexes.

4.4.5. Post-processing chain

A URI should always pass through this chain even if a decision not to crawl the URI was done in a processor earlier in the chain. The post-processing chain must contain the following processors (or replacement modules that perform similar operations):

- **CrawlStateUpdater**

  Updates the per-host information that may have been affected by the fetch. This is currently robots and IP address info.

- **LinksScoper**

  Checks all links extracted from the current download against the crawl scope. Those that are out of scope are discarded. Logging of discarded URLs can be enabled.

- **FrontierScheduler**

  'Schedules' any URLs stored as CandidateURIs found in the current CrawlURI with the frontier for crawling. Also schedules prerequisites if any.

5. Settings

The settings framework is designed to be a flexible way to configure a crawl with special treatment for subparts of the web without adding too much performance overhead. If you want to write a module which should be configurable through the user interface, it is important to have a basic understanding of the Settings framework (for the impatient, have a look at Section 6, “Common needs for all configurable modules” for code examples). At its core the settings framework is a way to keep persistent, context sensitive configuration settings for any class in the crawler.

All classes in the crawler that have configurable settings, subclass ComplexType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html] or one of its descendants. The ComplexType implements the javax.management.DynamicMBean interface. This gives you a way to ask the object for what attributes it supports the standard methods for getting and setting these attributes.

The entry point into the settings framework is the SettingsHandler [http://crawler.archive.org/apidocs/org/archive/crawler/settings/SettingsHandler.html]. This class is responsible for loading and saving from persistent storage and for interconnecting the different parts of the framework.

Figure 3. Schematic view of the Settings Framework

5.1. Settings hierarchy

The settings framework supports a hierarchy of settings. This hierarchy is built by CrawlerSettings [http://crawler.archive.org/apidocs/org/archive/crawler/settings/CrawlerSettings.html] objects. On the top there is a settings object representing the global settings. This consists of all the settings that a crawl job needs for running. Beneath this global object there is one "per" settings object for each host/domain which has settings that should override the order for that particular host or domain.

When the settings framework is asked for an attribute for a specific host, it will first try to see if this attribute is set for this particular host. If it is, the value will be returned. If not, it will go up one level recursively until it eventually reaches the order object and returns the global value. If no value is set here either (normally it would be), a hard coded default value is returned.

All per domain/host settings objects only contain those settings which are to be overridden for that particular domain/host. The convention is to name the top level object "global settings" and the objects beneath "per settings" or "overrides" (although the refinements described next, also do overriding).

To further complicate the picture, there is also settings objects called refinements. An object of this type belongs to a global or per settings object and overrides the settings in its owners object if some criteria is met. These criteria could be that the URI in question conforms to a regular expression or that the settings are consulted at a specific time of day limited by a time span.

5.2. ComplexType hierarchy

All the configurable modules in the crawler subclasses ComplexType or one of its descendants. The ComplexType is responsible for keeping the definition of the configurable attributes of the module. The actual values are stored in an instance of DataContainer. The DataContainer is never accessed directly from user code. Instead the user accesses the attributes through methods in the ComplexType. The attributes are accessed in different ways depending on if it is from the user interface or from inside a running crawl.

When an attribute is accessed from the URI (either reading or writing) you want to make sure that you are editing the attribute in the right context. When trying to override an attribute, you don't want the settings framework to traverse up to an effective value for the attribute, but instead want to know that the attribute is not set on this level. To achieve this, there is getLocalAttribute(CrawlerSettings settings, String name) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getLocalAttribute(org.archive.crawler.settings.CrawlerSettings,%20java.lang.String)] and setAttribute(CrawlerSettings settings, Attribute attribute) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#setAttribute(org.archive.crawler.settings.CrawlerSettings,%20javax.management.Attribute)] methods taking a settings object as a parameter. These methods works only on the supplied settings object. In addition the methods getAttribute(name) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getAttribute(java.lang.String)] and setAttribute(Attribute attribute) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#setAttribute(javax.management.Attribute)] is there for conformance to the Java JMX specification. The latter two always works on the global settings object.

Getting an attribute within a crawl is different in that you always want to get a value even if it is not set in its context. That means that the settings framework should work its way up the settings hierarchy to find the value in effect for the context. The method getAttribute(String name, CrawlURI uri) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getAttribute(java.lang.String,%20org.archive.crawler.datamodel.CrawlURI)] should be used to make sure that the right context is used. The Figure 4, “Flow of getting an attribute” shows how the settings framework finds the effective value given a context.

Figure 4. Flow of getting an attribute

The different attributes each have a type. The allowed types all subclass the Type [http://crawler.archive.org/apidocs/org/archive/crawler/settings/Type.html] class. There are tree main types:

1. SimpleType
2. ListType
3. ComplexType

Except for the SimpleType, the actual type used will be a subclass of one of these main types.

5.2.1. SimpleType

The SimpleType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/SimpleType.html] is mainly for representing Java wrappers for the Java primitive types. In addition it also handles the `java.util.Date` type and a special Heritrix TextField [http://crawler.archive.org/apidocs/org/archive/crawler/settings/TextField.html] type. Overrides of a SimpleType must be of the same type as the initial default value for the SimpleType.

5.2.2. ListType

The ListType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ListType.html] is further subclassed into versions for some of the wrapped Java primitive types (DoubleList [http://crawler.archive.org/apidocs/org/archive/crawler/settings/DoubleList.html], FloatList [http://crawler.archive.org/apidocs/org/archive/crawler/settings/FloatList.html], IntegerList [http://crawler.archive.org/apidocs/org/archive/crawler/settings/IntegerList.html], LongList [http://crawler.archive.org/apidocs/org/archive/crawler/settings/LongList.html], StringList [http://crawler.archive.org/apidocs/org/archive/crawler/settings/StringList.html]). A List holds values in the same order as they were added. If an attribute of type ListType is overridden, then the complete list of values is replaced at the override level.

5.2.3. ComplexType

The ComplexType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html] is a map of name/value pairs. The values can be any Type including new ComplexTypes. The ComplexType is defined abstract and you should use one of the subclasses MapType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/MapType.html] or ModuleType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ModuleType.html]. The MapType allows adding of new name/value pairs at runtime, while the ModuleType only allows the name/value pairs that it defines at construction time. When overriding the MapType the options are either override the value of an already existing attribute or add a new one. It is not possible in an override to remove an existing attribute. The ModuleType doesn't allow additions in overrides, but the predefined attributes' values might be overridden. Since the ModuleType is defined at construction time, it is possible to set more restrictions on each attribute than in the MapType. Another consequence of definition at construction time is that you would normally subclass the ModuleType, while the MapType is usable as it is. It is possible to restrict the MapType to only allow attributes of a certain type. There is also a restriction that MapTypes can not contain nested MapTypes.

6. Common needs for all configurable modules

As mentioned earlier all configurable modules in Heritrix subclasses ComplexType (or one of its descendants). When you write your own module you should inherit from ModuleType [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ModuleType.html] which is a subclass of ComplexType intended to be subclassed by all modules in Heritrix.

6.1. Definition of a module

Heritrix knows how to handle a ComplexType and to get the needed information to render the user interface part for it. To make this happen your module has to obey some rules.

1. A module should always implement a constructor taking exactly one argument - the name argument (see ModuleType(String name) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ModuleType.html#ModuleType(java.lang.String)]).

2. All attributes you want to be configurable should be defined in the constructor of the module.

6.1.1. The obligatory one argument constructor

All modules need to have a constructor taking a String argument. This string is used to identify the module. In the case where a module is of a type that is replacing an existing module of which there could only be one, it is important that the same name is being used. In this case the constructor might choose to ignore the name string and substitute it with a hard coded one. This is for example the case with the Frontier. The name of the Frontier should always be the string "frontier". For this reason the Frontier interface that all Frontiers should implement has a static variable:

```
public static final String ATTR_NAME = "frontier";
```

which implementations of the Frontier use instead of the string argument submitted to the constructor. Here is the part of the default Frontiers' constructor that shows how this should be done.

```
public Frontier(String name) {
    //The 'name' of all frontiers should be the same (Frontier.ATTR_NAME)
    //therefore we'll ignore the supplied parameter.
    super(Frontier.ATTR_NAME, "HostQueuesFrontier. Maintains the internal" +
        " state of the crawl. It dictates the order in which URIs" +
        " will be scheduled. \nThis frontier is mostly a breadth-first" +
        " frontier, which refrains from emitting more than one" +
        " CrawlURI of the same \'key\' (host) at once, and respects" +
        " minimum-delay and delay-factor specifications for" +
        " politeness.");
```

As shown in this example, the constructor must call the superclass's constructor. This example also shows how to set the description of a module. The description is used by the user interface to guide the user in configuring the crawl. If you don't want to set a description (strongly discouraged), the ModuleType also has a one argument constructor taking just the name.

6.1.2. Defining attributes

The attributes on a module you want to be configurable must be defined in the modules constructor. For this purpose the ComplexType has a method addElementToDefinition(Type type) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#addElementToDefinition(org.archive.crawler.settings.Type)]. The argument given to this method is a definition of the attribute. The Type [http://crawler.archive.org/apidocs/org/archive/crawler/settings/Type.html] class is the superclass of all the attribute definitions allowed for a ModuleType. Since the ComplexType, which ModuleType inherits, is itself a subclass of Type, you can add new ModuleTypes as attributes to your module. The Type class implements configuration methods common for all Types that defines an attribute on your module. The addElementToDefinition method returns the added Type so that it is easy to refine the configuration of the Type. Lets look at an example (also from the default Frontier) of an attribute definition.

```
public final static String ATTR_MAX_OVERALL_BANDWIDTH_USAGE =
        "total-bandwidth-usage-KB-sec";
private final static Integer DEFAULT_MAX_OVERALL_BANDWIDTH_USAGE =
        new Integer(0);
...

Type t;
t = addElementToDefinition(
    new SimpleType(ATTR_MAX_OVERALL_BANDWIDTH_USAGE,
    "The maximum average bandwidth the crawler is allowed to use. " +
    "The actual readspeed is not affected by this setting, it only " +
    "holds back new URIs from being processed when the bandwidth " +
    "usage has been to high.\n0 means no bandwidth limitation.",
    DEFAULT_MAX_OVERALL_BANDWIDTH_USAGE));
t.setOverrideable(false);
```

Here we add an attribute definition of the SimpleType (which is a subclass of Type). The SimpleType's constructor takes three arguments: name, description and a default value. Usually the name and default value are defined as constants like here, but this is of course optional. The line **t.setOverrideable(false);** informs the settings framework to not allow per overrides on this attribute. For a full list of methods for configuring a Type see the Type [http://crawler.archive.org/apidocs/org/archive/crawler/settings/Type.html] class.

6.2. Accessing attributes

In most cases when the module needs to access its own attributes, a CrawlURI is available. The right way to make sure that all the overrides and refinements are considered is then to use the method getAttribute(String name, CrawlURI uri) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getAttribute(java.lang.String,%20org.archive.crawler.datamodel.CrawlURI)] to get the attribute. Sometimes the context you are working in could be defined by other objects than the CrawlURI, then use the getAttribute(Object context, String name) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getAttribute(java.lang.Object,%20java.lang.String)] method to get the value. This method tries its best at getting some useful context information out of an object. What it does is checking if the context is any kind of URI or a settings object. If it can't find anything useful, the global settings are used as the context. If you don't have any context at all, which is the case in some initialization code, the getAttribute(String name) [http://crawler.archive.org/apidocs/org/archive/crawler/settings/ComplexType.html#getAttribute(java.lang.String)] could be used.

6.3. Putting together a simple module

From what we learned so far, let's put together a module that doesn't do anything useful, but show some of the concepts.

```
package myModule;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.management.AttributeNotFoundException;
import org.archive.crawler.settings.MapType;
import org.archive.crawler.settings.ModuleType;
import org.archive.crawler.settings.RegularExpressionConstraint;
import org.archive.crawler.settings.SimpleType;
import org.archive.crawler.settings.Type;

public class Foo extends ModuleType {
  private static Logger logger = Logger.getLogger("myModule.Foo");

  public Foo(String name) {
    Type mySimpleType1 = new SimpleType(
                "name1", "Description1", new Integer(10));
    addElementToDefinition(mySimpleType1);

    Type mySimpleType2 = new SimpleType(
                "name2", "Description2", "defaultValue");
    addElementToDefinition(mySimpleType2);
    mySimpleType2.addConstraint(new RegularExpressionConstraint(
                ".*Val.*", Level.WARNING,
                "This field must contain 'Val' as part of the string."));

    Type myMapType = new MapType("name3", "Description3", String.class);
    addElementToDefinition(myMapType);
  }

  public void getMyTypeValue(CrawlURI curi) {
    try {
      int maxBandwidthKB = ((Integer) getAttribute("name1", curi)).intValue();
    } catch (AttributeNotFoundException e) {
      logger.warning(e.getMessage());
    }
  }

  public void playWithMap(CrawlURI curi) {
    try {
      MapType myMapType = (MapType) getAttribute("name3", curi);
      myMapType.addElement(
              null, new SimpleType("name", "Description", "defaultValue"));
      myMapType.setAttribute(new Attribute("name", "newValue"));
    } catch (Exception e) {
      logger.warning(e.getMessage());
    }
  }
}
```

This example shows several things:

One thing that we have not mentioned before is how we do general error logging. Heritrix uses the standard Java 1.4 logging facility. The convention is to initialize it with the class name.

Here we define and add a SimpleType that takes an Integer as the argument and setting it to '10' as the default value.

It is possible to add constraints on fields. In addition to be constrained to only take strings, this field add a requirement that the string should contain 'Val' as part of the string. The constraint also has a level and a description. The description is used by the user interface to give the user a fairly good explanation if the submitted value doesn't fit in with the constraint. Three levels are honored. Level.INFO

Level.INFO — Values are accepted even if they don't fulfill the constraint's requirement. This is used when you don't want to disallow the value, but warn the user that the value seems to be out of reasonable bounds.

Level.WARNING — The value must be accepted by the constraint to be valid in crawl jobs, but is legal in profiles even if it doesn't. This is used to be able to put values into a profile that a user should change for every crawl job derived from the profile.

Level.SEVERE — The value is not allowed whatsoever if it isn't accepted by the constraint.

See the Constraint [http://crawler.archive.org/apidocs/org/archive/crawler/settings/Constraint.html] class for more information.

This line defines a MapType allowing only Strings as values.

An example of how to read an attribute.

Here we add a new element to the MapType. This element is valid for this map because its default value is a String.

Now we change the value of the newly added attribute. JMX requires that the new value is wrapped in an object of type Attribute which holds both the name and the new value.

Note

To make your module known to Heritrix, you need to make mention of it in the appropriate `src/conf/modules` file: i.e. if your module is a Processor, it needs to be mentioned in the `Processor.options` file. The options files get built into the Heritrix jar.

“A little known fact about Heritrix: When trying to read modules/Processor.options Heritrix will concatenate any such files it finds on the classpath. This means that if you write your own processor and wrap it in a jar you can simply include in that jar a modules/Processor.options file with just the one line needed to add your processor. Then simply add the new jar to the $HERITRIX_HOME/lib directory and you are done. No need to mess with the Heritrix binar-

> ies. For an example of how this is done, look at the code for this project: deduplicator [http://vefsofnun.bok.hi.is/deduplicator/index.html] ” [Kristinn Sigurðsson on the mailing list, 3281 [http://tech.groups.yahoo.com/group/archive-crawler/message/3281]].

If everything seems ok so far, then we are almost ready to write some real modules.

7. Some notes on the URI classes

URIs[1] in Heritrix are represented by several classes. The basic building block is `org.archive.datamodel.UURI` which subclasses `org.apache.commons.httpclient.URI`. "UURI" is an abbrieviation for "Usable URI." This class always normalizes and derelativizes URIs -- implying that if a UURI instance is successfully created, the represented URI will be, at least on its face, "usable" -- neither illegal nor including superficial variances which complicate later processing. It also provides methods for accessing the different parts of the URI.

We used to base all on `java.net.URI` but because of bugs and its strict RFC2396 conformance in the face of a real world that acts otherwise, its facility was was subsumed by UURI.

Two classes wrap the `UURI` in Heritrix:

CandidateURI [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CandidateURI.html]
: A URI, discovered or passed-in, that may be scheduled (and thus become a CrawlURI). Contains just the fields necessary to perform quick in-scope analysis. This class wraps an UURI instance.

CrawlURI [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html]
: Represents a candidate URI and the associated state it collects as it is crawled. The CrawlURI is an subclass of CandidateURI. It is instances of this class that is fed to the processors.

7.1. Supported Schemes (UnsupportedUriSchemeException)

A property in `heritrix.properties` named `org.archive.crawler.datamodel.UURIFactory.schemes` lists supported schemes. Any scheme not listed here will cause an UnsupportedUriSchemeException which will be reported in urierrors.log with a 'Unsupported scheme' prefix. If you add a fetcher for a scheme, you'll need to add to this list of supported schemes (Later, Heritrix can ask its fetchers what schemes are supported).

7.2. The CrawlURI's Attribute list

The CrawlURI offers a flexible attribute list which is used to keep arbitrary information about the URI while crawling. If you are going to write a processor you almost certainly will use the attribute list. The attribute list is a key/value-pair list accessed by typed accessors/setters. By convention the key values are picked from the CoreAttributeConstants [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CoreAttributeConstants.html] interface which all processors implement. If you use other keys than those listed in this interface, then you

[1] URI (Uniform Resource Identifiers) defined by RFC 2396 [http://www.ietf.org/rfc/rfc2396.txt] is a way of identifying resources. The relationship between URI, URL and URN is described in the RFC: ["A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource. The term "Uniform Resource Name" (URN) refers to the subset of URI that are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable."] Although Heritrix uses URIs, only URLs are supported at the moment. For more information on naming and addressing of resources see: Naming and Addressing: URIs, URLs, ... [http://www.w3.org/Addressing/] on w3.org's website.

must add the handling of that attribute to some other module as well.

7.3. The recorder streams

The crawler wouldn't be of much use if it did not give access to the HTTP request and response. For this purpose the CrawlURI has the getHttpRecorder() [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#getHttpRecorder()] [2] method. The recorder is referenced by the CrawlURI and available to all the processors. See Section 11, “Writing a Processor” for an explanation on how to use the recorder.

8. Writing a Frontier

As mentioned before, the Frontier is a pluggable module responsible for deciding which URI to process next, and when. The Frontier is also responsible for keeping track of other aspects of the crawls internal state which in turn can be used for logging and reporting. Even though the responsibilities of the Frontier might not look overwhelming, it is one of the hardest modules to write well. You should really investigate if your needs could not be met by the existing Frontier, or at least mostly met by subclassing an existing Frontier. With these warnings in mind, let's go ahead and create a really simple Frontier.

```
package mypackage;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import org.archive.crawler.datamodel.CandidateURI;
import org.archive.crawler.datamodel.CrawlURI;
import org.archive.crawler.datamodel.FetchStatusCodes;
import org.archive.crawler.datamodel.UURI;
import org.archive.crawler.framework.CrawlController;
import org.archive.crawler.framework.Frontier;
import org.archive.crawler.framework.FrontierMarker;
import org.archive.crawler.framework.exceptions.FatalConfigurationException;
import org.archive.crawler.framework.exceptions.InvalidFrontierMarkerException;
import org.archive.crawler.settings.ModuleType;


/**
 * A simple Frontier implementation for tutorial purposes
 */
public class MyFrontier extends ModuleType implements Frontier,
        FetchStatusCodes {
    // A list of the discovered URIs that should be crawled.
    List pendingURIs = new ArrayList();

    // A list of prerequisites that needs to be met before any other URI is
    // allowed to be crawled, e.g. DNS-lookups
    List prerequisites = new ArrayList();

    // A hash of already crawled URIs so that every URI is crawled only once.
    Map alreadyIncluded = new HashMap();

    // Reference to the CrawlController.
```

[2] This method will most likely change name see Section 1, “The org.archive.util.HTTPRecorder class” for details.

```
    CrawlController controller;

    // Flag to note if a URI is being processed.
    boolean uriInProcess = false;

    // top-level stats
    long successCount = 0;
    long failedCount = 0;
    long disregardedCount = 0;
    long totalProcessedBytes = 0;

    public MyFrontier(String name) {
        super(Frontier.ATTR_NAME, "A simple frontier.");
    }

    public void initialize(CrawlController controller)
            throws FatalConfigurationException, IOException {
        this.controller = controller;

        // Initialize the pending queue with the seeds
        this.controller.getScope().refreshSeeds();
        List seeds = this.controller.getScope().getSeedlist();
        synchronized(seeds) {
            for (Iterator i = seeds.iterator(); i.hasNext();) {
                UURI u = (UURI) i.next();
                CandidateURI caUri = new CandidateURI(u);
                caUri.setSeed();
                schedule(caUri);
            }
        }
    }

    public synchronized CrawlURI next(int timeout) throws InterruptedException {
        if (!uriInProcess && !isEmpty()) {
            uriInProcess = true;
            CrawlURI curi;
            if (!prerequisites.isEmpty()) {
                curi = CrawlURI.from((CandidateURI) prerequisites.remove(0));
            } else {
                curi = CrawlURI.from((CandidateURI) pendingURIs.remove(0));
            }
            curi.setServer(controller.getServerCache().getServerFor(curi));
            return curi;
        } else {
            wait(timeout);
            return null;
        }
    }

    public boolean isEmpty() {
        return pendingURIs.isEmpty() && prerequisites.isEmpty();
    }

    public synchronized void schedule(CandidateURI caURI) {
        // Schedule a uri for crawling if it is not already crawled
        if (!alreadyIncluded.containsKey(caURI.getURIString())) {
            if(caURI.needsImmediateScheduling()) {
                prerequisites.add(caURI);
            } else {
                pendingURIs.add(caURI);
            }
            alreadyIncluded.put(caURI.getURIString(), caURI);
        }
    }

    public void batchSchedule(CandidateURI caURI) {
        schedule(caURI);
    }

    public void batchFlush() {
    }

    public synchronized void finished(CrawlURI cURI) {
        uriInProcess = false;
        if (cURI.isSuccess()) {
            successCount++;
            totalProcessedBytes += cURI.getContentSize();
            controller.fireCrawledURISuccessfulEvent(cURI);
            cURI.stripToMinimal();
        } else if (cURI.getFetchStatus() == S_DEFERRED) {
            cURI.processingCleanup();
            alreadyIncluded.remove(cURI.getURIString());
            schedule(cURI);
        } else if (cURI.getFetchStatus() == S_ROBOTS_PRECLUDED
                || cURI.getFetchStatus() == S_OUT_OF_SCOPE
                || cURI.getFetchStatus() == S_BLOCKED_BY_USER
                || cURI.getFetchStatus() == S_TOO_MANY_EMBED_HOPS
                || cURI.getFetchStatus() == S_TOO_MANY_LINK_HOPS
                || cURI.getFetchStatus() == S_DELETED_BY_USER) {
            controller.fireCrawledURIDisregardEvent(cURI);
            disregardedCount++;
            cURI.stripToMinimal();
        } else {
            controller.fireCrawledURIFailureEvent(cURI);
            failedCount++;
            cURI.stripToMinimal();
        }
        cURI.processingCleanup();
    }

    public long discoveredUriCount() {
        return alreadyIncluded.size();
    }

    public long queuedUriCount() {
        return pendingURIs.size() + prerequisites.size();
    }

    public long finishedUriCount() {
        return successCount + failedCount + disregardedCount;
    }

    public long successfullyFetchedCount() {
        return successCount;
    }

    public long failedFetchCount() {
        return failedCount;
    }

    public long disregardedFetchCount() {
        return disregardedCount;
    }

    public long totalBytesWritten() {
        return totalProcessedBytes;
    }

    public String report() {
        return "This frontier does not return a report.";
    }

    public void importRecoverLog(String pathToLog) throws IOException {
        throw new UnsupportedOperationException();
    }

    public FrontierMarker getInitialMarker(String regexpr,
            boolean inCacheOnly) {
        return null;
    }

    public ArrayList getURIsList(FrontierMarker marker, int numberOfMatches,
            boolean verbose) throws InvalidFrontierMarkerException {
        return null;
    }

    public long deleteURIs(String match) {
        return 0;
    }

}
```

Note

To test this new Frontier you must add it to the classpath. Then to let the user interface be aware of it, you must add the fully qualified classname to the `Frontier.options` file in the `conf/modules` directory.

This Frontier hands out the URIs in the order they are discovered, one at a time. To make sure that the web servers are not overloaded it waits until a URI is finished processing before it hands out the next one. It does not retry URIs for other reasons than prerequisites not met (DNS lookup and fetching of robots.txt). This Frontier skips a lot of the tasks a real Frontier should take care of. The first thing is that it doesn't log anything. A real Frontier would log what happened to every URI. A real Frontier would also be aware of the fact that Heritrix is multi threaded and try to process as many URIs simultaneously as allowed by the number of threads without breaking the politeness rules. Take a look at Frontier [http://crawler.archive.org/xref/org/archive/crawler/frontier/BdbFrontier.html] (javadoc) [http://crawler.archive.org/apidocs/org/archive/crawler/frontier/BdbFrontier.html] to see how a full blown Frontier might look like.

All Frontiers must implement the Frontier interface. Most Frontiers will also implement the FetchStatusCodes because these codes are used to determine what to do with a URI after it has returned from the processing cycle. In addition you might want to implement the CrawlStatusListener [http://crawler.archive.org/apidocs/org/archive/crawler/event/CrawlStatusListener.html] which enables the Frontier to be aware of starts, stops, and pausing of a crawl. For this simple example we don't care about that. The most important methods in the Frontier interface are:

1. next(int timeout)
2. schedule(CandidateURI caURI)
3. finished(CrawlURI cURI)

The Figure 5, “Frontier data flow” shows a simplified sequence diagram of the Frontiers collaboration with other classes. For readability, the processors (of which there are more than showed in this diagram) are chained together in this diagram. It is actually the ToeThread that runs each processor in turn.

Figure 5. Frontier data flow

As the diagram shows, the next() method of the Frontier will return URI's from the prerequisite list before the pending queue is considered. Let's take a closer look at the implementation.

```
public synchronized CrawlURI next(int timeout) throws InterruptedException {
    if (!uriInProcess && !isEmpty()) {
        uriInProcess = true;
        CrawlURI curi;
        if (!prerequisites.isEmpty()) {
            curi = CrawlURI.from((CandidateURI) prerequisites.remove(0));
        } else {
            curi = CrawlURI.from((CandidateURI) pendingURIs.remove(0));
        }
        curi.setServer(controller.getServerCache().getServerFor(curi));
        return curi;
    } else {
        wait(timeout);
        return null;
    }
}
```

First we check if there is a URI in process already, then check if there are any URIs left to crawl.

Make sure that we let the prerequisites be processed before any regular pending URI. This ensures that DNS-lookups and fetching of robots.txt is done before any "real" data is fetched from the host. Note that DNS-lookups are treated as an ordinary URI from the Frontier's point of view. The next lines pulls a CandidateURI from the right list and turn it into a CrawlURI suitable for being crawled. The CrawlURI.from(CandidateURI) [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#from(org.archive.crawler.datamodel.CandidateURI)] method is used because the URI in the list might already be a CrawlURI and could then be used directly. This is the case for URIs where the preconditions was not met. As we will see further down these URIs are put back into the pending queue.

This line is very important. Before a CrawlURI can be processed it must be associated with a CrawlServer. The reason for this, among others, is to be able to check preconditions against the URI's host (for example so that DNS-lookups are done only once for each host, not for every URI).

In this simple example, we are not being aware of the fact that Heritrix is multithreaded. We just let the method wait the timeout time and the return null if no URIs where ready. The intention of the timeout is that if no URI could be handed out at this time, we should wait the timeout before returning null. But if a URI becomes available during this time it should wake up from the wait and hand it out. See the javadoc for next(timeout) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Frontier.html#next(int)] to get an explanation.

When a URI has been sent through the processor chain it ends up in the LinksScoper. All URIs should end up here even if the preconditions where not met and the fetching, extraction and writing to the archive has been postponed. The LinksScoper iterates through all new URIs (prerequisites and/or extracted URIs) added to the CrawlURI and, if they are within the scope, converts them from Link objects to CandidateURI objects. Later in the postprocessor chain, the FrontierScheduler adds them to Frontier by calling the schedule(CandidateURI) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Frontier.html#schedule(org.archive.crawler.datamodel.CandidateURI)] method. There is also a batch version of the schedule method for efficiency, see the javadoc [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Frontier.html] for more information. This simple Frontier treats them the same.

```
public synchronized void schedule(CandidateURI caURI) {
        // Schedule a uri for crawling if it is not already crawled
        if (!alreadyIncluded.containsKey(caURI.getURIString())) {
            if(caURI.needsImmediateScheduling()) {
                prerequisites.add(caURI);
            } else {
                pendingURIs.add(caURI);
            }
            alreadyIncluded.put(caURI.getURIString(), caURI);
        }
    }
```

This line checks if we already has scheduled this URI for crawling. This way no URI is crawled more than once.

If the URI is marked by a processor as a URI that needs immediate scheduling, it is added to the prerequisite queue.

Add the URI to the list of already scheduled URIs.

After all the processors are finished (including the FrontierScheduler's scheduling of new URIs), the ToeThread calls the Frontiers finished(CrawlURI) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Frontier.html#finished(org.archive.crawler.datamodel.CrawlURI)] method submitting the CrawlURI that was sent through the chain.

```
    public synchronized void finished(CrawlURI cURI) {
        uriInProcess = false;
        if (cURI.isSuccess()) {
            successCount++;
            totalProcessedBytes += cURI.getContentSize();
            controller.fireCrawledURISuccessfulEvent(cURI);
            cURI.stripToMinimal();
        } else if (cURI.getFetchStatus() == S_DEFERRED) {
            cURI.processingCleanup();
            alreadyIncluded.remove(cURI.getURIString());
            schedule(cURI);
        } else if (cURI.getFetchStatus() == S_ROBOTS_PRECLUDED
                || cURI.getFetchStatus() == S_OUT_OF_SCOPE
                || cURI.getFetchStatus() == S_BLOCKED_BY_USER
                || cURI.getFetchStatus() == S_TOO_MANY_EMBED_HOPS
                || cURI.getFetchStatus() == S_TOO_MANY_LINK_HOPS
                || cURI.getFetchStatus() == S_DELETED_BY_USER) {
            controller.fireCrawledURIDisregardEvent(cURI);
            disregardedCount++;
            cURI.stripToMinimal();
        } else {
            controller.fireCrawledURIFailureEvent(cURI);
            failedCount++;
            cURI.stripToMinimal();
        }
        cURI.processingCleanup();
    }
```

The processed URI will have status information attached to it. It is the task of the finished method to check these statuses and treat the URI according to that (see Section 2, “The Frontiers handling of dispositions”).

If the URI was successfully crawled we update some counters for statistical purposes and "forget about it".

Modules can register with the controller [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlController.html] to receive notifications [http://crawler.archive.org/apidocs/org/archive/crawler/event/CrawlURIDispositionListener.html] when decisions are made on how to handle a CrawlURI. For example the StatisticsTracker [http://crawler.archive.org/apidocs/org/archive/crawler/admin/StatisticsTracker.html] is dependent on these notifications to report the crawler's progress. Different fireEvent methods are called on the controller for each of the different actions taken on the CrawlURI.

We call the stripToMinimal method so that all data structures referenced by the URI are removed. This is done so that any class that might want to serialize the URI could be do this as efficient as possible.

If the URI was deferred because of a unsatisfied precondition, reschedule it. Also make sure it is removed from the already included map.

This method nulls out any state gathered during processing.

If the status is any of the one in this check, we treat it as disregarded. That is, the URI could be crawled, but we don't want it because it is outside some limit we have defined on the crawl.

If it isn't any of the previous states, then the crawling of this URI is regarded as failed. We notify about it and then forget it.

9. Writing a Filter

Filters[3] are modules that take a CrawlURI and determine if it matches the criteria of the filter. If so it returns true, otherwise it returns false.

A filter could be used in several places in the crawler. Most notably is the use of filters in the Scope. Aside that, filters are also used in processors. Filters applied to processors always filter URIs out. That is to say that any URI matching a filter on a processor will effectively skip over that processor. This can be useful to disable (for instance) link extraction on documents coming from a specific section of a given website.

All Filters should subclass the Filter [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Filter.html] class. Creating a filter is just a matter of implementing the innerAccepts(Object) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Filter.html#innerAccepts(java.lang.Object)] method. Because of the planned overhaul of the scopes and filters, we will not provide a extensive example of how to create a filter at this point. It should be pretty easy though to follow the directions in the javadoc. For your filter to show in the application interface, you'll need to edit `src/conf/modules/Filter.options`

10. Writing a Scope

A CrawlScope [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlScope.html] [3] instance defines which URIs are "in" a particular crawl. It is essentially a Filter which determines (actually it subclasses the Filter [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Filter.html] class), looking at the totality of information available about a CandidateURI/CrawlURI instance, if that URI should be scheduled for crawling. Dynamic information inherent in the discovery of the URI, such as the path by which it was discovered, may be considered. Dynamic information which requires the consultation of external and potentially volatile information, such as current robots.txt requests and the history of attempts to crawl the same URI, should NOT be considered. Those potentially high-latency decisions should be made at another step.

As with Filters, the scope will be going through a refactoring. Because of that we will only briefly describe how to create new Scopes at this point.

All Scopes should subclass the CrawlScope [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlScope.html] class. Instead of overriding the innerAccepts method as you would do if you created a filter, the CrawlScope class imple-

[3] It has been identified problems with how the Scopes are defined. Please see the user manual for a discussion of the problems with the current Scopes [http://crawler.archive.org/articles/user_manual/config.html#scopeproblems]. The proposed changes to the Scope will affect the Filters as well.

ments this and instead offers several other methods that should be overriden instead. These methods acts as different type of filters that the URI will be checked against. In addition the CrawlScope class offers a list of exclude filters which can be set on every scope. If the URI is accepted (matches the test) by any of the filters in the exclude list, it will be considered being out of scope. The implementation of the innerAccepts method in the CrawlSope is as follows:

```
protected final boolean innerAccepts(Object o) {
    return ((isSeed(o) || focusAccepts(o)) || additionalFocusAccepts(o) ||
            transitiveAccepts(o)) && !excludeAccepts(o);
}
```

The result is that the checked URI is considered being inside the crawl's scope if it is a seed or is accepted by any of the focusAccepts, additionalFocusAccepts or transitiveAccepts, unless it is matched by any of the exclude filters.

When writing your own scope the methods you might want to override are:

- focusAccepts(Object) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlScope.html#focusAccepts(java.lang.Object)] the focus filter should rule things in by prima facia/regexp-pattern analysis. The typical variants of the focus filter are:

    - broad: accept all

    - domain: accept if on same 'domain' (for some definition) as seeds

    - host: accept if on exact host as seeds

    - path: accept if on same host and a shared path-prefix as seeds

    Heritrix ships with scopes that implement each of these variants. An implementation of a new scope might thus be subclassing one of these scopes.

- additionalFocusAccepts(Object) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlScope.html#additionalFocusAccepts(java.lang.Object)]

- transitiveAccepts(Object) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/CrawlScope.html#transitiveAccepts(java.lang.Object)] the transitive filter rule extra items in by dynamic path analysis (for example, off site embedded images).

11. Writing a Processor

All processors extend org.archive.crawler.framework.Processor [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Processor.html]. In fact that is a complete class and could be used as a valid processor, only it doesn't actually do anything.

Extending classes need to override the innerProcess(CrawlURI) [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Processor.html#innerProcess(org.archive.crawler.datamodel.CrawlURI)] method on it to add custom behavior. This method will be invoked for each URI being processed and this is therfore where a processor can affect it.

Basically the innerProcess method uses the CrawlURI that is passed to it as a parameter (see Section 11.1, “Accessing and updating the CrawlURI”) and the underlying HttpRecorder (managed by the

ToeThread) (see Section 11.2, “The HttpRecorder”) to perform whatever actions are needed.

Fetchers read the CrawlURI, fetch the relevant document and write to the HttpRecorder. Extractors read the HttpRecorder and add the discovered URIs to the CrawlURIs list of discovered URIs etc. Not all processors need to make use of the HttpRecorder.

Several other methods can also be optionally overriden for special needs:

- initialTasks() [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Processor.html#initialTasks()]

  This method will be called after the crawl is set up, but before any URIs are crawled.

  Basically it is a place to write initialization code that only needs to be run once at the start of a crawl.

  Example: The FetchHTTP [http://crawler.archive.org/apidocs/org/archive/crawler/fetcher/FetchHTTP.html] processor uses this method to load the cookies file specified in the configuration among other things.

- finalTasks() [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Processor.html#innerProcess(org.archive.crawler.datamodel.CrawlURI)]

  This method will be called after the last URI has been processed that will be processed. That is at the very end of a crawl (assuming that the crawl terminates normally).

  Basically a place to write finalization code.

  Example: The FetchHTTP processor uses it to save cookies gathered during the crawl to a specified file.

- report() [http://crawler.archive.org/apidocs/org/archive/crawler/framework/Processor.html#innerProcess(org.archive.crawler.datamodel.CrawlURI)]

  This method returns a string that contains a human readable report on the status and/or progress of the processor. This report is accessible via the WUI and is also written to file at the end of a crawl.

  Generally, a processor would include the number of URIs that they've handled, the number of links extracted (for link extractors) or any other abitrary information of relevance to that processor.

11.1. Accessing and updating the CrawlURI

The CrawlURI contains quite a bit of data. For an exhaustive look refer to its Javadoc [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html].

- getAlist() [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#getAList()]

  This method returns the CrawlURI's 'AList'.

  ### Note

  `getAlist()` has been deprecated. Use the typed accessors/setters instead.
  The AList is basically a hash map. It is used instead of the Java HashMap or Hashtable because it is more efficient (especially when it comes to serializing it). It also has typed methods for adding and

getting strings, longs, dates etc.

Keys to values and objects placed in it are defined in the CoreAttributeConstants [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CoreAttributeConstants.html].

It is of course possible to add any arbitrary entry to it but that requires writing both the module that sets the value and the one that reads it. This may in fact be desirable in many cases, but where the keys defined in CoreAttributeConstants suffice and fit we strongly recommend using them.

The following is a quick overview of the most used CoreAttributeConstants:

- **A_CONTENT_TYPE**

  Extracted MIME type of fetched content; should be set immediately by fetching module if possible (rather than waiting for a later analyzer)

- **LINK COLLECTIONS**

  There are several Java Collection containing URIs extracted from different sources. Each link is a Link containing the extracted URI. The URI can be relative. The LinksScoper will read this list and convert Links inscope to CandidateURIs for adding to the Frontier by FrontierScheduler.

  ## Note

  CrawlURI has the convenience method addLinkToCollection(link, collection) [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#addLinkToCollection(java.lang.String,%20java.lang.String)] for adding links to these collections. This methods is the prefered way of adding to the collections.

  - *A_CSS_LINKS*

    URIs extracted from CSS stylesheets

  - *A_HTML_EMBEDS*

    URIs belived to be embeds.

  - *A_HTML_LINKS*

    Regularly discovered URIs. Despite the name the links could have (in theroy) been found

  - *A_HTML_SPECULATIVE_EMBEDS*

    URIs discovered via aggressive link extraction. Are treated as embeds but generally with lesser tolerance for nested embeds.

  - *A_HTTP_HEADER_URIS*

    URIs discovered in the returned HTTP header. Usually only redirects.

See the Javadoc [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CoreAttributeConstants.html] for CoreAttributeConstants for more.

- setHttpRecorder(HttpRecorder) [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#setHttpRecorder(org.archive.util.HttpRecorder)]

  Set the HttpRecorder that contains the fetched document. This is generally done by the fetching processor.

- getHttpRecorder() [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#setHttpRecorder(org.archive.util.HttpRecorder)]

  A method to get at the fetched document. See Section 11.2, “The HttpRecorder” for more.

- getContentSize() [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#getContentSize()]

  If a document has been fetched, this method will return its size in bytes.

- getContentType() [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html#getContentType()]

  The content (mime) type of the fetched document.

For more see the Javadoc [http://crawler.archive.org/apidocs/org/archive/crawler/datamodel/CrawlURI.html] for CrawlURI.

11.2. The HttpRecorder

A HttpRecorder [http://crawler.archive.org/apidocs/org/archive/util/HttpRecorder.html] is attached to each CrawlURI that is successfully fetched by the FetchHTTP [http://crawler.archive.org/apidocs/org/archive/crawler/fetcher/FetchHTTP.html] processor. Despite its name it could be used for non-http transactions if some care is taken. This class will likely be subject to some changes in the future to make it more general.

Basically it pairs together a RecordingInputStream and RecordingOutputStream to capture exactly a single HTTP transaction.

11.2.1. Writing to HttpRecorder

Before it can be written to a processor, it must get a reference to the current threads HttpRecorder. This is done by invoking the HttpRecorder class' static method getHttpRecorder() [http://crawler.archive.org/apidocs/org/archive/util/HttpRecorder.html#getHttpRecorder()]. This will return the HttpRecorder for the current thread. Fetchers should then add a reference to this to the CrawlURI via the method discussed above.

Once a processor has the HttpRecorder object it can access its RecordingInputStream [http://crawler.archive.org/apidocs/org/archive/io/RecordingInputStream.html] stream via the getRecordedInput() [http://crawler.archive.org/apidocs/org/archive/util/HttpRecorder.html#getRecordedInput()] method. The RecordingInputStream extends InputStream and should be used to capture the incoming document.

11.2.2. Reading from HttpRecorder

Processors interested in the contents of the HttpRecorder can get at its ReplayCharSequence [http://crawler.archive.org/apidocs/org/archive/io/ReplayCharSequence.html] via its getReplayCharSequence() [http://crawler.archive.org/apidocs/org/archive/util/HttpRecorder.html#getReplayCharSequence()] method. The ReplayCharSequence is basically a java.lang.CharSequence that can be read normally. As discussed above the CrawlURI has a method for getting at the existing HttpRecorder.

11.3. An example processor

The following example is a very simple extractor.

```
package org.archive.crawler.extractor;

import java.util.regex.Matcher;

import javax.management.AttributeNotFoundException;

import org.archive.crawler.datamodel.CoreAttributeConstants;
import org.archive.crawler.datamodel.CrawlURI;
import org.archive.crawler.framework.Processor;
import org.archive.crawler.settings.SimpleType;
import org.archive.crawler.settings.Type;
import org.archive.crawler.extractor.Link;
import org.archive.util.TextUtils;

/**
 * A very simple extractor. Will assume that any string that matches a
 * configurable regular expression is a link.
 *
 * @author Kristinn Sigurdsson
 */
public class SimpleExtractor extends Processor
    implements CoreAttributeConstants
{
    public static final String ATTR_REGULAR_EXPRESSION = "input-param";
    public static final String DEFAULT_REGULAR_EXPRESSION =
        "http://([a-zA-Z0-9]+\\.)+[a-zA-Z0-9]+/"; //Find domains

    int numberOfCURIsHandled = 0;
    int numberOfLinksExtracted = 0;

    public SimpleExtractor(String name) {
        super(name, "A very simple link extractor. Doesn't do anything useful.");
        Type e;
        e = addElementToDefinition(new SimpleType(ATTR_REGULAR_EXPRESSION,
            "How deep to look into files for URI strings, in bytes",
            DEFAULT_REGULAR_EXPRESSION));
        e.setExpertSetting(true);
    }

    protected void innerProcess(CrawlURI curi) {

        if (!curi.isHttpTransaction())
        {
            // We only handle HTTP at the moment.
            return;
        }

        numberOfCURIsHandled++;

        CharSequence cs = curi.getHttpRecorder().getReplayCharSequence();
        String regexpr = null;
        try {
            regexpr = (String)getAttribute(ATTR_REGULAR_EXPRESSION,curi);
        } catch(AttributeNotFoundException e) {
            regexpr = DEFAULT_REGULAR_EXPRESSION;
        }

        Matcher match = TextUtils.getMatcher(regexpr, cs);
        while (match.find()){
            String link = cs.subSequence(match.start(),match.end()).toString();
            curi.createAndAddLink(link, Link.SPECULATIVE_MISC, Link.NAVLINK_HOP);
            numberOfLinksExtracted++;
            System.out.println("SimpleExtractor: " + link);
        }

        TextUtils.recycleMatcher(match);
    }

    public String report() {
        StringBuffer ret = new StringBuffer();
        ret.append("Processor: org.archive.crawler.extractor." +
            "SimpleExtractor\n");
        ret.append("  Function:          Example extractor\n");
        ret.append("  CrawlURIs handled: " + numberOfCURIsHandled + "\n");
        ret.append("  Links extracted:   " + numberOfLinksExtracted + "\n\n");

        return ret.toString();
    }
}
```

The constructor. As with any Heritrix module it set's up the processors name, description and configurable parameters. In this case the only configurable parameter is the Regular expression that will be used to find links. Both a name and a default value is provided for this parameter. It is also marked as an expert setting.

Check if the URI was fetched via a HTTP transaction. If not it is probably a DNS lookup or was not fetched. Either way regular link extraction is not possible.

If we get this far then we have a URI that the processor will try to extract links from. Bump URI counter up by one.

Get the ReplayCharSequence. Can apply regular expressions on it directly.

Look up the regular expression to use. If the attribute is not found we'll use the default value.

Apply the regular expression. We'll use the TextUtils.getMatcher() [http://crawler.archive.org/apidocs/org/archive/util/TextUtils.html#getMatcher(java.lang.String,%20java.lang.CharSequence)] utility method for performance reasons.

Extract a link discovered by the regular expression from the character sequence and store it as a string.

Add discovered link to the collection of regular links extracted from the current URI.

Note that we just discovered another link.

This is a handy debug line that will print each extracted link to the standard output. You would not want this in production code.

Free up the matcher object. This too is for performance. See the related javadoc [http://crawler.archive.org/apidocs/org/archive/util/TextUtils.html#freeMatcher(java.util.regex.Matcher)].

The report states the name of the processor, its function and the totals of how many URIs were handled and how many links were extracted. A fairly typical report for an extractor.

Even though the example above is fairly simple the processor nevertheless works as intended.

11.4. Things to keep in mind when writing a processor

11.4.1. Interruptions

Classes extending Processor should not trap InterruptedExceptions.

InterruptedExceptions should be allowed to propagate to the ToeThread executing the processor.

Also they should immediately exit their main method (`innerProcess()`) if the `interrupted` flag is set.

11.4.2. One processor, many threads

For each processor only one instance is created per crawl. As there are multiple threads running, these processors must be carefully written so that no conflicts arise. This usually means that class variables can not be used for other things then gathering incremental statistics and data.

There is a facility for having an instance per thread but it has not been tested and will not be covered in this document.

12. Writing a Statistics Tracker

A Statistics Tracker is a module that monitors the crawl and records statistics of interest to it.

Statistics Trackers must implement the StatisticsTracking interface [http://crawler.archive.org/apidocs/org/archive/crawler/framework/StatisticsTracking.html]. The interface imposes very little on the module.

Its initialization method provides the new statistics tracker with a reference to the CrawlController and thus the module has access to any part of the crawl.

Generally statistics trackers gather information by either querying the data exposed by the Frontier or by listening for CrawlURI disposition events [http://crawler.archive.org/apidocs/org/archive/crawler/event/CrawlURIDispositionListener.html] and crawl status events [http://crawler.archive.org/apidocs/org/archive/crawler/event/CrawlStatusListener.html].

The interface extends Runnable. This is based on the assumptions that statistics tracker are proactive in gathering their information. The CrawlController will start each statistics tracker once the crawl begins. If this facility is not needed in a statistics tracker (i.e. all information is gathered passively) simply implement the run() method as an empty method.

Note

For new statistics tracking modules to be available in the web user interface their class name must be added to the `StatisticsTracking.options` file under the conf/modules directory. The classes' full name (with package info) should be written in its own line, followed by a '|' and a descriptive name (containing only [a-z,A-Z]).

12.1. AbstractTracker

A partial implementation of a StatisticsTracker is provided in the frameworks package. The AbstractTracker [http://crawler.archive.org/apidocs/org/archive/crawler/framework/AbstractTracker.html] implements the StatisticsTracking interface and adds the needed infrastructure for doing snapshots of the crawler status.

This is done implementing the thread aspects of the statistics tracker. This means that classes extending the AbstractTracker need not worry about thread handling, implementing the logActivity() method allows them to poll any information at fixed intervals.

AbstractTracker also listens for crawl status events and pauses and stops its activity based on them.

12.2. Provided StatisticsTracker

The admin package contains the only provided implementation of the statistics tracking interface. The StatisticsTracker [http://crawler.archive.org/apidocs/org/archive/crawler/admin/StatisticsTracker.html] is designed to write progress information to the progress-statistics.log as well as providing the web user interface with information about ongoing and completed crawls. It also dumps various reports at the end of each crawl.

13. Internet Archive ARC files

By default, heritrix writes all its crawled to disk using ARCWriterProcessor [http://crawler.archive.org/apidocs/org/archive/crawler/writer/ARCWriterProcessor.html]. This processor writes the found crawl content as Internet Archive ARC files. The ARC file format is described here: Arc File Format [http://www.archive.org/web/researcher/ArcFileFormat.php]. Heritrix writes version 1 ARC files.

By default, Heritrix writes *compressed* version 1 ARC files. The compression is done with gzip, but rather compress the ARC as a whole, instead, each ARC Record is in turn gzipped. All gzipped records are concatenated together to make up a file of multiple gzipped members. This concatenation, it turns out, is a legal gzip file; you can give it to gzip and it will undo each compressed record in turn. Its an amenable compression technique because it allows random seek to a single record and the undoing of that record only. Otherwise, first the total ARC would have to be uncompressed to get any one record.

Pre-release of Heritrix 1.0, an amendment was made to the ARC file version 1 format to allow writing of extra metadata into first record of an ARC file. This extra metadata is written as XML. The XML Schema used by metadata instance documents can be found at http://archive.org/arc/1.0/xsd [http://archive.org/arc/1.0/arc.xsd]. The schema is documented here [http://archive.org/arc/1.0/arc.html].

If the extra XML metadata info is present, the second '<reserved>' field of the second line of version 1 ARC files will be changed from '0' to '1': i.e. ARCs with XML metadata are version '1.1'.

If present, the ARC file metadata record body will contain at least the following fields (Later revisions to the ARC may add other fields):

1. Software and software version used creating the ARC file. Example: 'heritrix 0.7.1 http://crawler.archive.org'.
2. The IP of the host that created the ARC file. Example: '103.1.0.3'.
3. The hostname of the host that created the ARC file. Example: 'debord.archive.org'.
4. Contact name of the crawl operator. Default value is 'admin'.
5. The http-header 'user-agent' field from the crawl-job order file. This field is recorded here in the metadata only until the day ARCs record the HTTP request made. Example: 'os-heritrix/0.7.0 (+http://localhost.localdomain)'.
6. The http-header 'from' from the crawl-job order file. This field is recorded here in the metadata only until the day ARCs record the HTTP request made. Example: 'webmaster@localhost.localdomain'.
7. The 'description' from the crawl-job order file. Example: 'Heritrix integrated selftest'
8. The Robots honoring policy. Example: 'classic'.
9. Organization on whose behalf the operator is running the crawl. Example 'Internet Archive'.
10. The recipient of the crawl ARC resource if known. Example: 'Library of Congress'.

13.1. ARC File Naming

When heritrix creates ARC files, it uses the following template naming them:

```
        <OPERATOR SPECIFIED> '-' <12 DIGIT TIMESTAMP> '-' <SERIAL NO.> '-' <FQDN H
```

... where <OPERATOR SPECIFIED> is any operator specified text, <SERIAL NO> is a zero-padded 5 digit number and <FQDN HOSTNAME> is the fully-qualified domainname on which the crawl was run.

13.2. Reading arc files

ARCReader [http://crawler.archive.org/apidocs/org/archive/io/arc/ARCReader.html] can be used reading arc files. It has a command line interface that can be used to print out meta info in a pseudo CDX format [http://www.archive.org/web/researcher/example_cdx.php] and for doing random access getting of arc records (The command-line interface is described in the main method javadoc [http://crawler.archive.org/apidocs/org/archive/io/arc/ARCReader.html#main(java.lang.String[])] comments).

Netarchive.dk [http://netarchive.dk/website/sources/index-en.php] have also developed arc reading and writing tools.

Tom Emerson of Basis Technology has put up a project on sourceforge to host a BSD-Licensed C++ ARC reader called libarc [http://sourceforge.net/projects/libarc/] (Its since been moved to archive-access [http://archive-access.sourceforge.net/]).

The French National Library (BnF) has also released a GPL perl/c ARC Reader. See BAT [http://crawler.archive.org/cgi-bin/wiki.pl?BnfArcTools] for documentation and where to download.. See Hedaern [http://archive-access.cvs.sourceforge.net/viewcvs.py/archive-access/archive-access/projects/hedaern/] for python readers/writers and for a skeleton webapp that allows querying by timestamp+date as well as full-text search of ARC content..

13.3. Writing arc files

Here is an example arc writer application: Nedlib To ARC conversion [http://nwatoolset.sourceforge.net/docs/NedlibToARC/]. It rewrites nedlib [http://www.csc.fi/sovellus/nedlib/index.phtml.en] files as arcs.

13.4. Searching ARCS

Check out the NutchWAX [http://archive-access.sourceforge.net/projects/nutch/]+WERA [http://archive-access.sourceforge.net/projects/wera/] bundle.

A. Future changes in the API

This appendix lists issues in the API that needs investigation and eventually changes to be made. Read this so that you could avoid locking yourself to dependencies that is likely to change.

1. The org.archive.util.HTTPRecorder class

The intention is to allow Heritrix to handle all relevant protocols, not only http/https. The naming of HTTPRecorder indicates that it is only for HTTP. Hopefully it is not. We have not investigated it fully yet, but it might be that changing the name of the class is all there is to be done.

2. The Frontiers handling of dispositions

When the Frontier decides which URIs should be treated as deferred and which has failed, it must check the fetch status and have knowledge about every possible status code. This complicates adding new status codes. It should be possible to constrain the number of status groups the Frontier has to know and then let the processors decide which group a new status code falls into. Possible status groups are:

successful

failure

disregard

retry immediately

retry after any prerequisite

retry at some later time

B. Version and Release Numbering

Heritrix uses a version numbering scheme modeled after the one used for Linux kernels. Versions are 3 numbers:

[major] .[minor/mode] .[patchlevel]

The major version number, currently at one, increments upon significant architectural changes or the achievement of important milestones in capabilities. The minor/mode version number increments as progress is made within a major version, with the added constraint that all external releases have an even minor/mode version number, and all internal/development versions have an odd minor/mode version number.

The patchlevel number increments for small sets of changes, providing the most fine-grain time line of software evolution. Patchlevels increment regularly for internal/development (odd minor level) work, but only increment for external releases when an official update to the previous release version has been tested and packaged.

Version numbers are applied as tags of the form "heritrix-#_#_#". Branches occur at major transitions and are labeled with the form "heritrix_#_#'.

When a particular development-version is thought appropriate to become an external/"st able" release, it is considered a "Release Candidate" (No where is this written in versions). If testing confirms it is suitable for release, it is assigned the next even minor/mode value (and a zero patchlevel), version-labelled, and packaged for release. Immediately after release, and before additional coding occurs, the HEAD is assigned the next odd minor/mode value (and a zero patchlevel) in project/source files.

If patches are required to a released version, before the next release is ready, they are applied to a branch from the release version tag, tested, and released as the subsequent patchlevel.

Keep in mind that each version number is an integer, not merely a decimal digit. To use an extreme example: development version 2.99.99 would be followed by either the 2.99.100 development version patchlevel or the 2.100.0 release. (And after such a release, the next development version would be

2.101.0.)

C. Making a Heritrix Release

Before initiating a release, its assumed that the current HEAD or BRANCH version has been run through the integration self test, that all unit tests pass, that the embryonic test plan [http://crawler.archive.org/cgi-bin/wiki.pl?CrawlTestPlan] has been exercised, and that general usage shows HEAD -- or BRANCH -- to be release worthy.

1. Send a mail to the list to freeze commits until the all-clear is given.
2. Up the project.xml 'currentVersion' element value (See Version and Release Numbering [#release_numbering] for guidance on what version number to use)
3. Update releasenotes.xml bugs and RFEs closed since last release (Looking at source code history to figure what's changed is too hard; All major changes should have associated BUG and RFE).
4. Add news of the new release to the site main home page.
5. Generate the site. Review all documentation making sure it remains applicable. Fix at least the embarrassing. Make issues to have all that remains addressed.
6. Update the README.txt (We used to include in README text-only version of dependencies list, release notes including items fixed but now we just point to the pertintent html).
7. Commit all changes made above all in the one commit with a log message about new release. Commit files with the new version, the README.txt, home page, and all changes in documentation including the changelog additions.
8. Wait on a cruisecontrol successful build of all just committed. Download the src and binary latest builds from under the cruisecontrol 'build artifacts' link.
9. Build the cruisecontrol produced src distribution version.
10. Run both the binary and src-built product through the integration self test suite: % $HERITRIX_HOME/bin/heritrix --selftest
11. Tag the repository. If a bugfix release -- an increase in the least significant digits -- then tag the release. Otherwise, if a major or minor release make a branch (See Appendix B, *Version and Release Numbering* for more on release versioning). Tags should be named for the release name as in if the release is 1.10.2, then the tag should be heritrix-1_10_2 (You can't use dots in tag names). If a branch, then name the branch for the minor (or major) version name as in, if making a release 1.10.0, then name the branch heritrix-1_10 and if making 2.0.0, name the branch heritrix-2.
12. Update the project.xml 'currentVersion' and build.xml 'version' property to both be a version number beyond that of the release currently being made (If we're releasing 0.2.0, then increment to 0.3.0).
13. Login and upload the maven 'dist' product to sourceforge into the admin->File releases section.
14. Send announcement to mailinglist -- and give an all-clear that commits may resume -- and update our release state on freshmeat site (Here is the URL I used creating our freshmeat project: http://freshmeat.net/add-project/all-done/43820/46804/ -- 46804 is our project ID).

D. Settings XML Schema

The XML Schema that describes the crawler job order file can be viewed as xhtml here, heritrix_settings.html [https://archive-crawler.svn.sourceforge.net/svnroot/archive-crawler/trunk/heritrix/src/webapps/admin/heritrix_settings.html].

E. Profiling Heritrix

Heritrix uses JProfiler [http://www.ej-technologies.com/products/jprofiler/overview.html]. See [1002336] Figure what profiler to use [https://sourceforge.net/tracker/index.php?func=detail&aid=1002336&group_id=73833&atid=539102] for a (rambling) log of experience with the various profilers and of how we arrived at this choice.

Bibliography

[Heritrix User Guide] *Heritrix User Guide [http://crawler.archive.org/user.html]*. Internet Archive [http://www.archive.org].

[Sun Code Conventions] *Code Conventions for the Java Programming Language [http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html]*. Sun Microsystems, Inc..

[Java Programming Style Guidelines] *Java Programming Style Guidelines [http://geosoft.no/javastyle.html]*. Geotechnical Software Services.

[Towards web-scale web archeology] *Towards web-scale web archeology*. *Research Report 174 [http://citeseer.ist.psu.edu/leung01towards.html]*. Compaq Systems Research Center, Palo Alto, CA. September 10, 2001. S. Leung, S. Perl, R. Stata, and J. Wiener.

[On High-Performance Web Crawling.] *On High-Performance Web Crawling*. *Research Report 173 [http://citeseer.ist.psu.edu/najork01highperformance.html]*. Compaq Systems Research Center, Palo Alto, CA. September 26, 2001. Marc Najork and Allan Heydon.

[Mercator: A Scalable, Extensible Web Crawler] *Mercator: A Scalable, Extensible Web Crawler [http://citeseer.ist.psu.edu/heydon99mercator.html]*. Compaq Systems Research Center, Palo Alto, CA. 1999. Marc Najork and Allan Heydon.