NAME

noid − nice opaque identifier generator commands

SYNOPSIS

noid [**−f** *Dbdir*] [**−vh**] *Command Arguments*

DESCRIPTION

The **noid** utility creates minters (identifier generators) and accepts commands that operate them. Once created, a minter can be used to produce persistent, globally unique names for documents, databases, images, vocabulary terms, etc. Properly managed, these identifiers can be used as long term durable information object references within naming schemes such as ARK, PURL, URN, DOI, and LSID. At the same time, alternative minters can be set up to produce short-lived names for transaction identifiers, compact web server session keys, and other ephemera.

A **noid** minter is a lightweight database designed for efficiently generating, tracking, and binding unique identifiers, which are produced without replacement in random or sequential order, and with or without a check character that can be used for detecting transcription errors. A minter can bind identifiers to arbitrary element names and element values that are either stored or produced upon retrieval from rule-based transformations of requested identifiers, the latter having application in identifier resolution. Noid minters are very fast, scalable, easy to create and tear down, and have a relatively small footprint. They use BerkeleyDB as the underlying database.

An identifier generated by a **noid** minter is also known generically as a ‘‘noid’’ (standing for nice opaque identifier and rhyming with void). While a minter can record and bind any identifiers that you bring to its attention, often it is used to generate, bringing to your attention, identifier strings that carry no widely recognizable meaning. This semantic opaqueness reduces their vulnerability to era− and language-specific change, and helps persistence by making for identifiers that can age and travel well.

The form, number, and intended longevity of a minter’s identifiers are given by a Template and a Term supplied when the generator database is created. A supplied Term of ‘‘long’’ establishes extra restrictions and logging appropriate for the support of persistent identifiers. Across successive minting operations, the generator ‘‘uses up’’ its namespace (the pool of identifiers it is capable of minting) such that no identifier will ever be generated twice unless the supplied Term is ‘‘short’’ and the namespace is finite and completely exhausted. The default Term is ‘‘medium’’.

The **noid** utility parameters — flags, *Dbdir* (database location), *Command*, *Arguments* — are described later under COMMANDS AND MODES. There are also sections covering persistence, templates, rule-based mapping, URL interface, and name resolution.

TUTORIAL INTRODUCTION

Once the noid utility is installed, the command,

```
noid dbcreate s.zd
```

will create a minter for an unlimited number of identifiers. It produces a generator for medium term identifiers (the default) with the Template, `s.zd`, governing the order, number, and form of minted identifier strings. These identifiers will begin with the

constant part `s` and end in a digit (the final `d`), all within an unbounded sequential (`z`) namespace. The TEMPLATES section gives a full explanation. This generator will mint the identifiers, in order,

```
s0, s1, s2, ..., s9, s10, ..., s99, s100, ...
```

and never run out. To mint the first ten identifiers,

```
noid mint 10
```

When you're done, on a UNIX platform you can remove that minter with

```
rm -fr NOID
```

Now let's create a more complex minter.

```
noid dbcreate f5.reedeedk long 13030 cdlib.org oac/cmp
```

This produces a generator for long term identifiers that begin with the constant part `13030/f5`. Exactly 70,728,100 identifiers will be minted before running out.

The 13030 parameter is the registered Name Assigning Authority Number (NAAN) for the assigning authority known as ''cdlib.org'', and ''oac/cmp'' is a string chosen by the person setting up this minter to identify the project that will be operating it. This particular minter generates identifiers that start with the prefix `f5` in the 13030 namespace. If long term information retention is within the mission of your organization (this includes national and university libraries and archives), you may register for a globally unique NAAN by sending email to ark at cdlib dot org.

Identifiers will emerge in ''quasi−random'' order, each consisting of six characters matching up one-for-one with the letters `eedeed`.

```
noid mint 1
```

The first identifier should be `13030/f54x54g11`, with the namespace ranging from a low of `13030/f5000000s` to a high of `13030/f5zz9zz94`. You can create a ''locations'' element under a noid and bind three URLs to it with the command,

```
noid bind set 13030/f54x54g11 locations \
  'http://a.b.org/foo|http://c.d.org/bar|http://e.f.org/zaf'
```

The template's final `k` causes a computed check character to be added to the end of every generated identifier. It also accounts for why the lowest and highest noids look a little odd on the end. The final check character allows detection of the most common transcription errors, namely, incorrect entry of one character and the transposition of two characters. The next command takes three identifiers that someone might ask you about and determines that, despite appearances, only the first is in the namespace of this minter.

```
noid validate - 13030/f54x54g11 13030/f54y54g11 \
  13030/f54x45g11
```

To make way for creation of another minter, you can move the entire minter into a subdirectory with the command,

```
mkdir f57 ; mv NOID f57
```

A minter may be set up on a web server, allowing the NAA organization easily to

distribute name assigment to trusted parties operating from remote locations. The URL INTERFACE section describes the procedure in detail. Once set up, you could mint one identifier by entering a URL such as the following into your web browser:

```
http://foo.ucop.edu/nd/noidu_f57?mint+1
```

Using a different procedure, you can also make your identifier bindings (e.g., location information) visible to the Internet via a few web server configuration directives. The NAME RESOLUTION section explains this further.

IDENTIFIER – AN ASSOCIATION SUPPORTED BY BINDINGS

An identifier is not a string of character data — an identifier is an association between a string of data and an object. This abstraction is necessary because without it a string is just data. It's nonsense to talk about a string's breaking, or about its being strong, maintained, and authentic. But as a representative of an association, a string can do, metaphorically, the things that we expect of it.

Without regard to whether an object is physical, digital, or conceptual, to identify it is to claim an association between it and a representative string, such as ''Jane'' or ''ISBN 0596000278''. What gives your claim credibility is a set of verifiable assertions, or metadata, about the object, such as age, height, title, or number of pages. Verifiability is outside the scope of the noid utility, but you can use a minter to record assertions supporting an association by binding arbitrary named elements and values to the identifier. Noid database elements can be up to 4 gigabytes in length, and one noid minter is capable of recording billions of identifiers.

You don't have to use the noid binding features at all if you prefer to keep track of your metadata elsewhere, such as in a separate database management system (DBMS) or on a sheet of paper. In any case, for each noid generated, the minter automatically stores its own lightweight ''circulation'' record asserting who generated it and when. If most of your metadata is maintained in a separate database, the minter's own records play a back up role, providing a small amount of redundancy that may be useful in reconstructing database bindings that have become damaged.

An arbitrary database system can complement a noid minter without any awareness or dependency on noids. On computers, identifier bindings are typically managed using methods that at some point map identifier strings to database records and/or to filesystem entries (effectively using the filesystem as a DBMS). The structures and logistics for bindings maintenance may reside entirely with the minter database, entirely outside the minter database, or anywhere in between. An individual organization defines whatever maintenance configuration suits it.

PERSISTENCE

A persistent identifier is an identifier that an organization commits to retain in perpetuity. Associations, the *sine qua non* of identifiers, last only as long as they (in particular, their bindings) are maintained. Often maintaining identifiers goes hand in hand with controlling the objects to which they are bound. No technology exists that automatically manages objects and associations; persistence is a matter of service commitment, tools that support that commitment, and information that allows users receiving identifiers to make the best judgment regarding an organization's ability and intention to maintain them.

It will be normal for organizations to maintain their own assertions about identifiers that you issue, and vice versa. In general there is nothing to prevent discrepancies among sets of assertions. Effectively, the association — the identifier — is in the eye of the beholder. As a simple example, authors create bibliography entries for cited works, and in that process they make their claims, often with small errors, about such things as the author and title of the identified thing. It is common for a provider of an identifier-driven service such as digital object retrieval to allow users to review its own, typically better-maintained sets of identifier assertions (i.e., metadata), even if it minted none of the identifiers that it services. We call such an organization a Name Mapping Authority (NMA) because it ''maps'' identifiers to services. It is possible for an NMA to service identifiers even if it neither hosts nor controls any objects.

It will also be normal for archiving organizations to maintain their own peculiar ideas about what persistence means. Different flavors will exist even within one organization, where, for example, it may be appropriate to apply corrections to persistent objects of one category, to never change objects of another, and to remove objects of a third category with a promise never to re-assign those objects' identifiers. One institution will guarantee persistence for certain things, while the strongest commitment made by some prominent archives will be ''reasonable effort''. Given the range of possibilities, a memory organization will need to record not only the identities but also the support policies for objects in its care. Any database, including a noid minter, can be used for this purpose.

For persistence across decades or centuries, opinions regarding an object's identity and commitments made to various copies of it will tend naturally to become more diverse. An object may have been inherited through a chain of stewardship, subtle identity changes, and peaks of renewed interest stretching back to a completely unrelated and now defunct organization that created and named it. For its original identifier to have persisted across the intervening years, it must look the same as when first minted. At that particular time, global uniqueness required the minted identifier to bear the imprint of the issuing organization (the NAA, or Name Assigning Authority), which long ago ceased to have any responsibility for its persistence. There is thus no conflict in a mapping authority (NMA) servicing identifiers that originate in many different assigning authorities.

These notions of flavors of persistence and separation of name authority function are built into the ARK (Archival Resource Key) identifier scheme that the **noid** utility was partly created to support. By design, noid minters also work within other schemes in recognition that persistence has nothing to do with identifier syntax. Opaque identifiers can be used by any application needing to reduce the liability created when identifier strings contain linguistic fragments that, however appropriate or even meaningless they are today, may one day create confusion, give offense, or infringe on a trademark as the semantic environment around us and our communities evolves. If employed for persistence, noid minters ease the unavoidable costs of long term maintenance by having a small technical footprint and by being implemented completely as open source software. For more information on ARKs, please see <http://ark.cdlib.org/> .

COMMANDS AND MODES

Once again, the overall utility summary is

noid [**−f** *Dbdir*] [**−vh**] *Command Arguments*

In all invocations, output is intended to be both human− and machine−readable. Batch operations are possible, allowing multiple minting and binding commands within one invocation. In particular, if *Command* is given as a ‘‘−’’ argument, then actual *Commands* are read in bulk from the standard input.

The string, *Dbdir*, specifies the directory where the database resides. To protect database coherence, it should not be located on a filesystem such as NFS or AFS that doesn't support POSIX file locking semantics. *Dbdir* may be given with the **NOID** environment variable, overridable with the **−f** option. If those strings are empty, the name or link name of the **noid** executable (argv[0] for C programmers) is checked to see if it reveals *Dbdir*. If that check (described next) fails, *Dbdir* is taken to be the current directory.

To check the name of the executable for *Dbdir*, the final pathname component (tail) is examined and split at the first ‘‘_’’ encountered. If none, the check fails. Otherwise, the check is considered successful and the latter half is taken as naming *Dbdir* relative to the current directory. This mechanism is designed for cases when it is inconvenient to specify *Dbdir* (such as in the URL interface) or when you are running several minters at once. As an example, */usr/bin/noid_fk9* specifies a *Dbdir* of *fk9*.

All files associated with a minter will be organized in a subdirectory, *NOID*, of *Dbdir*; this has the consequence that there can be at most one minter in a directory. To allow **noid** to create a new minter in a directory already containing a *NOID* subdirectory, remove or rename the entire *NOID* subdirectory.

The **noid** utility may be run as a URL-driven web server application, such as in a CGI that allows name assignment via remote operator. If the executable begins **noidu...**, the noid URL mode is in effect. Input parameters, separated by a ‘‘+’’ sign, are expected to arrive embedded in the query part of a URL, and output will be formatted for display on an ordinary web browser. An executable of **noidu_xk4**, for example, would turn on URL mode and set *Dbdir* to *xk4*. This is further described under URL INTERFACE.

The **noid** utility may be run as a name resolver running behind a web server. If the executable begins **noidr...**, the noid resolver mode is in effect, which means that commands will be read from the standard input (as if only the ‘‘−’’ argument had been given) and the script output will be unbuffered. This mode is designed for machine interaction and is intended to be operated by rewriting rules listed in a web server configuration file as described later under NAME RESOLUTION AND REDIRECTION INTERFACE.

At minter creation time, a report summarizing its properties is produced and stored in the file, *NOID/README*. This report may be useful to the organization articulating the operating policy of the minter. In a formal context, such as the creation of a minter for long term identifiers, that organization is the Name Assigning Authority.

The **−v** option prints the current version of the **noid** utility and **−h** prints a help message.

In the *Command* list below, capitalized symbols indicate values to be replaced by the caller. Optional arguments are in [brackets] and (A|B|C) means one of A or B or C.

noid dbcreate [*Template* [*Term* [*NAAN NAA SubNAA*]]]

Create a database that will mint (generate) identifiers according to the given Template and Term. As a side-effect this causes the creation of a directory, *NOID*, within

Dbdir. If you have several generators, it may be convenient to operate each from within a *Dbdir* that uniquely identifies each Template; for example, you might change to a directory that you named *fk6* after the Template `fk.rdeedde` (‘‘fk’’ followed by 6 variable characters) of the minter that resides there.

The Term declares whether the identifiers are intended to be ‘‘long’’, ‘‘medium’’ (the default), or ‘‘short’’. A short term identifier minter is the only one that will re-mint identifiers after the namespace is exhausted, simply returning the oldest previously minted identifier. As mentioned earlier, however, some namespaces are unbounded and never run out of identifiers.

If Term is ‘‘long’’, the arguments NAAN, NAA, and SubNAA are required, and all minted identifiers will be returned with the NAAN and a ‘‘/’’ prepended to them. The NAAN is a namespace identifier and should be a globally unique Name Assigning Authority (NAA) number. Apply for one by email to ark@cdlib.org, or for testing purposes, use ‘‘00000’’ as a non-unique NAAN.

The NAA argument is the character string equivalent for the NAAN; for example, 13960 corresponds to the NAA, ‘‘archive.org’’. The SubNAA argument is also a character string, but is a locally determined and possibly structured subauthority string (e.g., ‘‘oac’’, ‘‘ucb/dpg’’, ‘‘practice_area’’) that is not globally registered.

If Template is not supplied, the minter freely binds any identifier that you submit without validating it first. In this case it also mints medium term identifiers under the default Template, `.zd`.

noid mint *N* [*Element Value*]

Generate N identifiers. If other arguments are specified, for each generated noid, add the given Element and bind it to the given Value. [Element−Value binding upon minting is not implemented yet.]

There is no ‘‘unmint’’ command. Once an identifier has been circulated in the outside world, it may be hard to withdraw because external users and systems will have bound it with their own assertions. Even within the minting organization, removing all of the identifier’s supporting bindings could entail actions such as file deletion that are outside the scope of the minter. While there is no command capable of withdrawing a circulated identifier, it is nonetheless easy to **queue** an identifier for reminting and to **hold** it against the possibility of minting at all. Identifiers that are long term should be treated as non-renewable resources except when you are absolutely sure about recycling them.

noid peppermint *N* [*Element Value*]

[This command is not implemented yet.] Generate N ‘‘peppered’’ identifiers. A peppered identifier is a regular identifier concatenated with a ‘‘!’’ character and a randomly generated cookie — the pepper — which serves as a kind of per-identifier password. (Salt is a technical term for some extra data that makes it harder to crack encrypted values; we use pepper for some extra data that makes it harder to crack unencrypted values.) To provide an extra level of database security, the base identifier, which is everything up to the ‘‘!’’, should be used in all public

communication, but the complete peppered identifier is required for all noid operations that would change values in the database.

As with the **mint** command, if other arguments are specified, for each generated noid, add the given Element and bind it to the given Value.

noid bind *How Id Element Value*

For the given Id, bind the Element to Value according to How. The Element and Value may be arbitrary strings. There are two reserved Element names allowing Values to be entered that are too large or syntactically inconvenient (depending on the calling environment's quoting restrictions) to pass in as command-line tokens.

If the Element is ''**:**'' and no Value is present, lines are read from the standard input up to a blank line; they will contain Element-colon-Value pairs in essentially email header format, with long values continued on indented lines. If the Element is ''**:−**'' and no Value is present, lines are read from the standard input up to end−of−file; the first non−comment, non-blank line must have an Element-colon to specify an Element name, and all the remaining input (up to EOF) is taken as its corresponding Value. Lines beginning with ''#'' are considered ''comment'' lines and are skipped.

The *How* argument specifies one of the following kinds of binding. Of these, the **set**, **add**, **insert**, and **purge** kinds ''don't care'' if there is no current binding.

new Only if Element does not exist, create a new binding.

replace
Only if Element exists, undo any old bindings and create a new binding.

set Means **new** or, failing that, **replace**.

append
Only if Element exists, place Value at the end of the old binding.

add Means **new** or, failing that, **append**.

prepend
Only if Element exists, place Value at the beginning of the old binding.

insert Means **new** or, failing that, **prepend**.

delete Remove any trace of Element, returning an error if it did not exist to begin with.

purge Remove any trace of Element, returning success whether or not it existed to begin with.

mint Means **new**, but ignore the Id argument (actually, confirm that it was given as **new**) and mint a new Id first.

peppermint
[This kind of binding is not implemented yet.] Means **new**, but ignore the Id argument (**new**) and peppermint a new Id first.

The RULE−BASED MAPPING section explains how to set up retrieval using non-stored values.

noid fetch *Id* [*Element* ...]

For the noid, Id, print with labels all bindings for the given Elements. If no Element is given, find and print all bindings for the given Id. This is the verbose version of the **get** command, in that it prints headers and labels for everything it finds.

noid get *Id* [*Element* ...]

For the noid, Id, print without labels all bindings for the given Elements. If no Element is given, find and print all bindings for the given Id. This is the quiet version of the **fetch** command, in that it suppresses all headers and labels. Between each Element requested, the output will be separated by a blank line.

noid hold (**set**|**release**) *Id* ...

Place or remove a **hold** on one or more Ids. A hold placed on an Id that has not been minted will cause it to be skipped when its turn to be minted comes around. A hold placed on an Id that has been minted will make it impossible to queue (typically for recycling). Minters of long term identifiers automatically place a hold on every minted noid. Holds can be placed or removed manually at any time.

noid queue (**now**|**first**|**lvf**|*Time*) *Id* ...

Queue one or more Ids for minting. Time is a number followed by units, which can be **d** for days or **s** for seconds (the default units). This can be used to recycle noids **now** or after a delay period. With **first**, the Id(s) will be queued such that they will be minted before any of the time-delayed entries. With **lvf** (Lowest Value First), the lowest valued identifier (intended for use with numeric identifiers) will be taken from the queue for minting before all others. [needs testing]

noid validate (*Template*|−) *Id* ...

Validate one or more Ids against a given Template, which, if given as ‘‘−’’, causes the minter’s native *Template* to be used.

TEMPLATES

A Template is a coded string of the form Prefix.Mask that is given to the noid **dbcreate** command to govern how identifiers will be minted. The Prefix, which may be empty, specifies an initial constant string. For example, upon database creation, in the Template

```
tb7r.zdd
```

the Prefix says that every minted identifier will begin with the literal string `tb7r`. Each identifier will end in at least two digits (`dd`), and because of the `z` they will be sequentially generated without limit. Beyond the first 100 mint operations, more digits will be added as needed. The minted noids will be, in order,

```
tb7r00, tb7r01, ..., tb7r100, tb7r101, ..., tb7r1000, ...
```

The period (‘.’) in the Template does not appear in the identifiers but serves to separate the constant first part (Prefix) from the variable second part (Mask). In the Mask, the first letter determines either random or sequential ordering and the remaining letters each match up with characters in a generated identifier. Perhaps the best way to introduce templates is with a series of increasingly complex examples.

`.rddd` to mint random 3−digit numbers, stopping after 1000th

`.sdddddd`	to mint sequential 6−digit numbers, stopping after millionth
`.zd`	sequential numbers without limit, adding new digits as needed
`bc.rdddd`	random 4−digit numbers with constant prefix `bc`
`8rf.sdd`	sequential 2−digit numbers with constant prefix `8rf`
`.se`	sequential extended-digits (from 0123456789bcdfghjkmnpqrstvwxz)
`h9.reee`	random 3−extended−digit numbers with constant prefix `h9`
`.zeee`	unlimited sequential numbers with at least 3 extended-digits
`.rdedeedd`	random 7−char numbers, extended-digits at chars 2, 4, and 5
`.zedededе`	unlimited mixed digits, adding new extended-digits as needed
`sdd.sdede`	sequential 4−mixed−digit numbers with constant prefix `sdd`
`.rdedk`	random 3 mixed digits plus final (4th) computed check character
`.sdeeedk`	5 sequential mixed digits plus final extended-digit check char
`.zdeek`	sequential digits plus check char, new digits added as needed
`63q.redek`	prefix plus random 4 mixed digits, one of them a check char

The first letter of the Mask, the generator type, determines the order and boundedness of the namespace. For example, in the Template `.sddd`, the Prefix is empty and the `s` says that the namespace is sequentially generated but bounded. The generator type may be one of,

`r` for quasi-randomly generated identifiers,

`s` for sequentially generated identifiers, limited in length and number by the length of the Mask,

`z` for sequentially generated identifiers, unlimited in length or number, re-using the most significant mask character (the second character of the Mask) as needed.

Although the order of minting is not obvious for `r` type minters, it is ‘‘quasi−random’’ in the sense that on your machine a minter created with the same Template will always produce the same sequence of noids over its lifetime. Quasi-random is a shade more predictable than pseudo-random (which, techically, is as random as computers get). This is a feature designed to help noid managers in case they are forced to start minting again from scratch; they simply process their objects over in the same order as before to recover the original assignments.

After the generator type, the rest of the Mask determines the form of the non-Prefix part, matching up letter-for-character with each generated noid character (an exception for the `z` case is described below). In the case of the Template `xv.sdddd`, the last four `d` Mask characters say that all identifiers will end with four digits, so the last identifier in the namespace is `xv9999`.

When `z` is used, the namespace is unbounded and therefore identifiers will eventually need to grow in length. To accommodate the growth, the second character (`e` or `d`) of the Mask will be repeated as often as needed; for instance, when all 4−digit numbers are used up, a 5th digit will be added. After the generator type character, Mask characters have the

following meanings:

`d` a pure digit, one of { 0123456789 }

`e` an ‘‘extended digit’’, one of { 0123456789bcdfghjkmnpqrstvwxz } (lower case only)

`k` a computed extended digit check character; if present, it must be the final Mask character

The set of extended digits is designed to help create more compact noids (a larger namespace for the same length of identifier) and discourage ‘‘accidental semantics’’, namely, the introduction of strings that have unintended but commonly recognized meanings. Opaque identifiers are desirable in many situations and the absence of vowels in extended digits is a step in that direction. To reduce visual mismatches, there is also no letter ‘‘l’’ (ell) because it is often mistaken for the digit ‘‘1’’.

The optional `k` Mask character, which may only appear at the end, enables detection of cases when a single character is mistyped and when two adjacent characters have been transposed — the most common transcription errors. A final `k` in the Mask will cause a check character to be appended after first computing it on the entire identifier generated so far, including the NAAN if one was specified at database creation time. For example, the final digit `1` in

```
13030/f54x54g11
```

was first computed over the string `13030/f54x54g1` and then added to the end.

RULE−BASED MAPPING

Any Element may be bound to a class of Ids such that retrieval against that Element for any Id in the class returns a computed value when no stored value exists. The class of Ids is specified via a regular expression (Perl−style) that will be checked for a match against Ids submitted via a retrieval operation (**get** or **fetch**) that names any Element bound in this manner. If the match succeeds, the element Value that was bound with the Id class is used as the right-hand side of a Perl substitution, and the resulting transformation is returned. We call this rule-based mapping, and it is probably best explained by working through the examples below.

To set up rule-based mapping for an Id class, construct a **bind** operation with an Id of the form `:idmap/`*Idpattern*, where *Idpattern* is a Perl regular expression. Then choose an Element name that you wish to have trigger the pattern match check whenever that Element is requested via a retrieval operation and a stored value does NOT exist; any Element will work as long as you use it for both binding and retrieving. Finally, specify a Value to be used as replacement text that transforms matching Ids into computed values via a Perl s/// substitution. As a simple example,

```
noid bind set :idmap/^ft redirect g7h
```

would cause any subsequent retrieval request against the Element named ‘‘redirect’’ to try pattern matching when no stored value is found. If the Id begins with ‘‘ft’’, it would then try to replace the ‘‘ft’’ with ‘‘g7h’’ and return the result as if it were a stored value. So if the Id were `ft89xr2t`, the command

```
noid get ft89xr2t redirect
```

would return `g7h89xr2t`. Fancier substitutions are possible, including replacement patterns that reference subexpressions in the original matching *Idpattern*. For example, the second command below,

```
noid bind set ':idmap/^ft([^x]+)x(.*)' my_elem '$2/g7h/$1'
noid get ft89xr2t my_elem
```

would return `r2t/g7h/89`. For ease of implementation, internally this kind of binding is stored and reported (which can be confusing) as the special noid, `:idmap/`*Element*, under element name *Idpattern*.

URL INTERFACE

Any number of minters can be operated behind a web server from a browser or any tool that activates URLs. This section describes a one-time set up procedure to make your server aware of minters, followed by another set up procedure for each minter. The one-time procedure involves creating a directory in your web server document tree where you will place one or more noid minter databases. In this example, the directory is *htdocs/nd* and we'll assume the **noid** script was originally installed in */usr/local/bin*.

```
mkdir htdocs/nd
cp -p /usr/local/bin/noid htdocs/nd/
```

The second command above creates an executable copy of the noid script that will be linked to for each minter you intend to expose to the web. To make your server recognize such links, include the line

```
ScriptAliasMatch ^/nd/noidu(.*) "/srv/www/htdocs/nd/noidu$1"
```

in your server configuration file and restart the server before trying the commands that follow. If you did not install the supporting *Noid.pm* module normally, you may also have to store a copy of it next to the script. This completes the one-time server set up.

Thereafter, for each minter that you wish to expose, it must first be allowed to write to its own database when invoked via the web server. Because it will be running under a special user at that time, before you create it, first become the user that your server runs under. In this example that user is ''wwwrun''.

```
cd htdocs/nd
su wwwrun
noid dbcreate kt.reeded
mkdir kt5
mv NOID kt5/
ln noid noidu_kt5
```

The third command above creates a minter for noids beginning with `kt` followed by 5 characters. The minter is then moved into its own directory within *htdocs/nd*. Finally, the last command makes a hard link (not a soft link) to the noid script, which for this minter will be invoked under the name **noidu_kt5**.

The URL interface is similar to the command line interface, but *Commands* are passed in via the query string of a URL where by convention a plus sign (''+'') is used instead of spaces to separate arguments. You will likely want to set up access restrictions (e.g., with an *.htaccess* file) so that only the parties you designate can generate identifiers. There is

also no **dbcreate** command available from the URL interface.

To mint one identifier, you could enter the following URL into your web browser, but replace ‘‘foo.ucop.edu’’ with your server’s name:

```
http://foo.ucop.edu/nd/noidu_kt5?mint+1
```

Reload to mint again. If you change the 1 to 20, you get twenty new and different noids.

```
http://foo.ucop.edu/nd/noidu_kt5?mint+20
```

To bind some data to an element called ‘‘myGoto’’ under one of the noids already minted,

```
http://foo.ucop.edu/nd/noidu_kt5?
    bind+set+13030/kt639k9+myGoto+http://foo.ucsd.edu/
```

In this case we stored a URL in ‘‘myGoto’’. This kind of convention can underly a redirection mechanism that is part of an organization’s overall identifier resolution strategy. To retrieve that stored data,

```
http://foo.ucop.edu/nd/noidu_kt5?get+13030/kt639k9+myGoto
```

Bulk operations can be performed over the web by invoking the URL with a query string of just ‘‘−’’, which will cause the minter to look for noid commands, one per line, in the POST data part of the HTTP request. If you put noid commands in a file *myCommands* and run the Unix utility

```
curl --data-binary @myCommands \
  'http://foo.ucop.edu/nd/noidu_kt5?-'
```

you could, for example, change the ‘‘myGoto’’ bindings for 500 noids in that one shell command. The output from each command in the file will be separated from the next (on the standard output) by a blank line.

NAME RESOLUTION AND REDIRECTION INTERFACE

In a URI context, *name resolution* is a computation, sometimes multi−stage, that translates a name into associated information of a particular type, often another name or an address. A *resolver* is a system that can perform one or more stages of a resolution. Noid minters can be set up as resolvers.

In our case, we’re interested in automatically translating access requests for each of a number of identifiers into requests for another kind of identifier. This is one tool in the persistent access strategy for naming schemes such as URL, ARK, PURL, Handle, DOI, and URN. You can use a noid minter to bind a second name to each identifier, even to identifiers that the minter did not generate. In principle, this will work with names from any scheme.

With web browsers, a central mechanism for name resolution is known as the server redirect, and mainstream web servers can easily be configured to redirect a half million different names without suffering in performance. You might choose not to use native web server redirects if you require resolution of several million names, or if you require software and infrastructure for non-URL-based names. Whatever your choice, maintaining a table that maps the first name to the second is an unavoidable burden.

As with the URL interface, any number of resolvers (minters underneath) can be operated

behind a web server from a browser or a tool that activates URLs. This section describes a one-time set up procedure to make your server aware of resolvers, followed by another set up procedure for each resolver. The one-time procedure involves creating a directory in your web server document tree where you will place one or more noid resolver databases. In this example (and in the previous example), we use *htdocs/nd*:

```
mkdir htdocs/nd
cp -p /usr/local/bin/noid htdocs/nd/
```

The second command above creates an executable copy of the noid script that will be linked to for each resolver you intend to expose. To make your server recognize such links, include the line (this is slightly different from the similar line in the previous section),

```
ScriptAliasMatch ^/nd/noidr(.*) "/srv/www/htdocs/nd/noidr$1"
```

in your server configuration file. If you did not install the supporting *Noid.pm* module normally, you may also have to store a copy of it next to the script. Then include the following lines in the configuration file; they form the start of a rewriting rule section that you will add to later for each resolver that you set up.

```
RewriteEngine on
# These next two files and their containing
# directory should be owned by "wwwrun".
RewriteLock   /var/log/rewrite/lock
RewriteLog    /var/log/rewrite/log
## RewriteLogLevel 9
```

The non-comment lines above initialize the rewriting system, identify the lock file used to synchronize access to the resolver, and identify the log file which can help in finalizing the exact rewrite rules that you use; disable logging with the default RewriteLogLevel value of 0, or set it as high as 9, with higher numbers producing more detailed information. This completes the one-time server set up for resolvers.

Thereafter, for each resolver that you wish to run, you need to set up a noid database and create a link of the form **noidr...** so that the noid script can be invoked in resolution mode. Unlike the URL interface, the resolution interface does not itself mint from the underlying minter. A separate URL interface may still be set up to mint and bind identifiers in the resolver database, or minting and binding can take place off the net.

In what follows, we will assume that you have set up a noid database with the same location and template as in the previous section. As before, the server is assumed to run under the user ‘‘wwwrun’’ and the database resides in *htdocs/nd/kt5*. As if our intentions included persistent identification, the minter in this example is for generating long term identifiers.

```
cd htdocs/nd
noid dbcreate kt.reeded long 13030 cdlib.org dpg
mkdir kt5
mv NOID kt5/
ln noid noidr_kt5
```

The last command makes a new hard link (not a soft link) to the noid script, which for this resolver will be invoked under the name **noidr_kt5**. The resolution interface is not called by a URL directly, but is invoked once upon server startup, where the **noidr...** prefix tells it to run in resolution mode. In this mode it loops, waiting for and responding to individual resolution attempts from the server itself.

To set up an individual resolver, define a Rewrite Map followed by a set of Rewrite Rules. This is done using server configuration file lines as shown in the next example. As with any change to the file, you will need to restart the server before it will have the desired effect.

```
# External resolution; start program once on server start
RewriteMap  rslv          prg:/srv/www/htdocs/nd/noidr_kt5
# Main lookup; add artificial prefix for subsequent testing
RewriteRule ^/ark:/(13030/.*)$ "_rslv_${rslv:get $1 myGoto}"

# Test: redirect [R] if it looks like a redirect
RewriteRule ^_rslv_([^:]*://.*)$    $1 [R]
# Test: strip prefix; pass through [PT] if intended for us
RewriteRule ^_rslv_(/.*)$           $1 [PT]
# Test: restore value if lookup failed; let come what may
RewriteRule ^_rslv_$                %{REQUEST_URI}
# Alternative: redirect failed lookup to a global resolver
```

When a request received by the server matches a Rewrite Rule, an attempt to resolve it via the running **noidr...** script is made. In this example, we will need to have bound a string representing a URL to the value for the fixed element name ‘‘myGoto’’ under each identifier that we wish to be resolvable. Building on the example from the previous section, assume the element ‘‘myGoto’’ holds the same URL as before for the noid `13030/kt639k9`. A browser retrieval request made by entering or clicking on

```
http://foo.ucop.edu/ark:/13030/kt639k9
```

would then result in a server redirect to

```
http://foo.ucsd.edu/
```

The resolution result for an identifier is whatever the **get** returns, which could as easily have retrieved a stored value as a rule-based value (allowing you to redirect many similar identifiers with one rule).

This approach to resolution does not address resolver discovery. An identifier found in the wild need not easily reveal whether it is actionable or resolvable, let alone which system or resolver to ask. The usual strategy for contemporary (web era) identifier schemes relies on well−known, scheme-dependent resolvers and web proxying of identifiers embedded in URLs. For example, global resolution for a non-proxied URN or Handle uses an undisclosed internet address, hard-coded into the resolver program, from which to start the resolution process. An ARK, PURL, or proxied Handle or URN tend to rely on a disclosed starting point. Whatever method is used for discovery, a noid resolver can in principle be used to resolve identifiers from any scheme.

NOID CHECK DIGIT ALGORITHM

The following describes the Noid Check Digit Algorithm (NCDA). Digits in question are actually ‘‘extended digits’’, or *xdigits*, which form an ordered set of *R* digits and characters. This set has radix *R*. In the examples below, we use a specific set of *R=29* xdigits.

When applied to substrings of well-formed identifiers, where the length of the substring is less than *R*, the NCDA is ‘‘perfect’’ for single digit and transposition errors, by far the most common user transcription errors (see David Bressoud, Stan Wagon, ‘‘Computational Number Theory, 2000, Key College Publishing’’). The NCDA is complemented by well-formedness rules that confirm the placement of constant data, including fixed labels and any characters that are not extended digits. After running the NCDA on the selected substring, the resulting check digit, an xdigit actually, is used either for comparing with a received check digit or for appending to the substring prior to issuing the identifier that will contain it.

For the algorithm to work, the substring in question must be less than *R* characters. The extended digit set used in the current instance is a sequence of *R=29* printable characters defined as follows:

```
xdigit:  0  1  2  3  4  5  6  7  8  9  b  c  d  f  g
 value:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14

xdigit:  h  j  k  m  n  p  q  r  s  t  v  w  x  z
 value: 15 16 17 18 19 20 21 22 23 24 25 26 27 28
```

Each xdigit in the identifier has the corresponding ordinal value shown. Any character not in the xdigit set is considered in the algorithm to have an ordinal value of zero.

A check digit is an xdigit computed from the base substring and then appended to form the ‘‘checked substring’’ (less than *R+1* characters long). To determine if a received identifier has been corrupted by a single digit or transposition error, the relevant substring is extracted and its last character is compared to the result of the same computation performed on the preceding substring characters.

The computation has two steps. Consider a base substring (no check digit appended) such as

```
13030/xf93gt2          (base substring)
```

Step 1. Check that the substring is well−formed, that is, that all non-xdigit characters (often constant character data) are exactly where expected; if not, the substring is not well-formed and the computation aborts. (This step is required to accommodate characters such as ‘‘/’’ that contribute nothing to the overall computation.)

Step 2. Multiply each character’s ordinal value by its position number (starting at position 1), and sum the products. For example,

```
 char: 1   3   0   3   0   /   x   f   9   3   g   t   2
  ord: 1   3   0   3   0   0  27  13   9   3  14  24   2
  pos: 1   2   3   4   5   6   7   8   9  10  11  12  13
 prod: 1 + 6 + 0 +12 + 0 + 0+189+104 +81 +30+154+288 +26=891
```

Step 3. The check digit is the xdigit whose ordinal value is that sum modulo *R* (divide

the sum by *R* and take the remainder).

In the example, *891 = 21* mod *R* (29) and so the check digit is `q`. This is appended to obtain the ‘‘checked substring’’, which is

`13030/xf93gt2q` (substring with check digit appended)

What follows is a two-part proof that this algorithm is ‘‘perfect’’ with respect to single digit and transposition errors.

Lemma 1: The NCDA is guaranteed against single-character errors.

Proof: We must prove that if two strings differ in one single character, then the check digit (xdigit) also differs. If the *n*−th xdigit’s ordinal is *d* in one string and *e* in another, the sums of products differ only by

```
(... + nd + ...) - (... + ne + ...) = n(d - e)
```

The check digits differ only if *n(d − e)* is not *0* mod *R*. Assume (contrapositively) that *n(d − e)* does equal *0* mod *R*. First, we know that *n(d − e)* is not zero because *n* is positive and *d* is different from *e*. Therefore, there must be at least one positive integer *i* such that

```
n(d - e) = Ri     =>     (n/i)(d - e) = R
```

Now, because *R* is prime,

```
either   (a)  n/i = 1    and   d - e = R
or       (b)  n/i = R    and   d - e = 1
```

But (a) cannot hold because xdigit ordinals differ by at most *R−1*. This leaves (b), which implies that there is an integer *i = n/R*. But since *R* is prime and *n* (a position number) is a positive integer less than *R*, then *0 < i < 1*, which cannot be true. So the check digits must differ.

Lemma 2: The NCDA is guaranteed against transposition of two single characters.

Proof: Non-contributing characters (non−xdigits) transposed with other characters will be detected in Step 1 when checking the constraints for well-formedness (e.g., the ‘‘/’’ must be at position 6 and only at position 6). Therefore we need only consider transposition of two xdigit characters. We must prove that if one string has an xdigit of ordinal *e* in position *i* and an xdigit of ordinal *d* in position *j*, and if another string is the same except for having *d* in position *i* and *e* in position *j*, then the check digits also differ. The sums of the products differ by

```
(... + ie + ... + jd + ...) - (... + id + ... + je + ...)
    = (ie + jd) - (id + je) = e(i - j) + d(j - i)
    =  d(j - i) - e(j - i)  = n(d - e)
```

where *n = j − i > 0* and *n < R*. The check digits differ only if *n(d − e) = 0* mod *R*. This reduces to the central statement of Lemma 1, which has been proven.

TO DO

Add features that are documented but not implemented yet: Element-Value binding upon minting; the **peppermint** command. The **append** and **prepend** kinds of binding currently have string-level semantics (new data is added as characters to an existing

element); should there also be list-level semantics (new data added as an extra subelement)?

Add extra options for **dbcreate**. An option to specify one or more identifier labels to strip from requests, and one canonical label to add upon minting and reporting. An option to set the initial seed for quasi-random ordering. Utilize the granular BerkeleyDB transaction and locking protection mechanisms.

Extend the Template Mask to allow for other character repertoires with prime numbers of elements. These would trade a some eye-friendliness for much more compact identifiers (cf. UUID/GUID), possibly also a way of asking that the last character of the repertoire only appear in the check character (e.g., for i and x below).

```
{ 0-9 x }                        cardinality 11, mask char i
{ 0-9 a-f _ }                    cardinality 17, mask char x
{ 0-9 a-z _ }                    cardinality 37, mask char v
{ 1-9 b-z B-Z } - {l, vowels}    cardinality 47, mask char E
{ 0-9 a-z A-Z # * + @ _ }        cardinality 67, mask char w
Visible ASCII - { % - . / \ }    cardinality 89, mask char c
```

Add support for simple file management associated with identifiers. For example, minting (and reminting) the noid `xv8t984` could result in the creation (and re−creation) of a corresponding canonical directory `xv/8t/98/4/`.

BETA SOFTWARE

This utility is in the beta phase of development. It is open source software written in the Perl scripting language with strictest type, value, and security checking enabled. While its readiness for long term application is still being evaluated, it comes with a growing suite of regression tests (currently about 250).

COPYRIGHT AND LICENSE

BUGS

Under case-insensitive file systems (e.g., Mac OS X), there is a chance for conflicts between the directory name *NOID*, script name *noid*, and module documentation requested (via perldoc) as *Noid*.

Not yet platform−independent.

Please report bugs to jak at ucop dot edu.

FILES

NOID	directory containing all database files related to a minter
NOID/noid.bdb	the BerkeleyDB database file at the heart of a minter
NOID/README	the creation record containing minter analysis details

SEE ALSO

dbopen (3), *perl* (1), *uuidgen* (1), <http://www.cdlib.org/inside/diglib/ark/>

AUTHORS

John A. Kunze, Michael A. Russell

PREREQUISITES

Perl Modules: Noid, BerkeleyDB, Config, Text::ParseWords, Getopt::Long, Fcntl, Sys::Hostname

Script Categories:

CGI UNIX : System_administration Web