XML::Comma Guide

Table of Contents

Table of Contents

Introduction

XML::Comma is an information management platform. Comma speeds the development of content−heavy, networked applications, and was designed to solve some of the problems that make managing extremely large web sites so expensive, difficult and tedious.

Comma is written mostly in Perl, and its target demographic is the Perl programmer who must build customized, complex systems that handle very large amounts of dynamic content. Like most software that is designed to be used by programmers to build other software, Comma is several things at once: a code library, a design framework, a development methodology and a runtime system all rolled into one. However, Comma's central philosophy is "play well with others," and the system depends heavily on a number of tools −− the Apache web server and its mod_perl extensions, the HTML::Mason web development environment, relational databases, the underlying filesystem and OS utilities −− to implement its functionality and to provide programmers with a complete, flexible, scalable, and familiar toolkit.

Comma shapes information into "documents," and −− as its (full) name implies −− uses XML to structure those documents. XML, like Perl, is a powerful and standard tool for organizing text. But XML, again like Perl, doesn't do much of anything by itself. Comma defines a number of discrete "processes" in the "life−cycle" of a document and provides a framework that abstracts basic activities common to those process. These frameworks include structuring and validation; long−term storage; programmatic manipulation; and indexing for fast sorting, categorization and retrieval.

This document, the XML::Comma User's Guide, is available in both HTML and PDF form. The PDF is generated from the HTML by HTMLDOC.

- HTML: http://xml−comma.org/guide−filter.html
- PDF: http://xml−comma.org/guide.pdf

Installation

Dependencies

XML::Comma requires that Perl, a number of CPAN modules, and a relational database be installed in order to function properly. The Perl version must be 5.005 or greater. The basic required CPAN modules (more may be used by additional parts of Comma) are **Proc::ProcessTable**, **DBI**, the **DBD::** module that matches your database and the **Digest::** module of your choice. The database can by mySQL or postgreSQL, with support for other databases available whenever someone asks for it.

Comma installs in the usual `make`, `make test` and `make install` fashion. The tests, however, won't run until the `Comma.pm` file has been edited to configure a number of standard variables to values that are appropriate for your system. `Comma.pm` controls the overall system configuration, and the version that is in the build directory will be copied to the appropriate location on your machine during the `make install` operation.

`Comma.pm` contains a hash named `$config`, the keys of which are used by the Comma internals to access the system configuration variables. Here is a list of those variables and a short description of each:

Comma.pm $config Variables

- **document_root** −− This is the default directory below which Comma documents will be found. This can actually be overridden in each store definition, but you will usually rely on this to be the base directory for document storage. The permissions on this directory must be set so that each user of the system has the read or write access that they will need to retrieve or store documents.
- **tmp_directory** −− This directory is available to the Comma system, and to any Comma code, for temporary storage. Most folks use `/tmp`.
- **defs_directories** −− This is an anonymous array of directories that contain document definitions and macros. Each directory must be explicitly listed; Comma does not recurse into subdirectories looking for definitions. The directories are searched in the order that they occur in this list (which is relevant only if you are worrying about naming collisions; definition loading happens infrequently enough that speed is not a concern).
- **defs_extension** −− The extension that Comma expects definition files to have. The convention is `.def`, which means that the *Foo* def will be found in a file called `Foo.def`.
- **macro_extension** −− Like **defs_extension**, only for macros. The convention is `.macro`.
- **parser** −− The parser module that Comma will use. The two best choices are `PurePerl` and `SimpleC`. The PurePerl module is written entirely in Perl, so should work on any system and without any installation headaches. The SimpleC module is faster, uses Brian Ingerson's really nifty Inline framework, and like all such things May Not Work for You. See the notes below.
- **hash_module** −− Comma generates checksums for documents. These checksums are used internally by the system, and are also available via the Doc **get_hash()** method. You can use any module that adheres to the CPAN "Digest" interface. `Digest::MD5` is a good choice.
- **sql_syntax** −− Comma needs to know which database you're using. The currently−supported options are `mysql` and `Pg`.
- **dbi_connect_info** −− This is an anonymous array containing the list of values that should be passed to the **DBI−>connect()** method when Comma attempts to open a connection to your relational database. Here is an example that could be used with mySQL:

```
dbi_connect_info => [ 'DBI:mysql:comma:localhost',
                                'user',
                                'password',
                                 { RaiseError => 1,
                                   PrintError => 0 } ],
```

- **log_file** -- This is the file that Comma will log errors and warnings to. Comma logging is relatively simple, and a value like `/tmp/comma.log` isn't a bad choice.

Using the SimpleC Parser

The SimpleC parser module requires that the Inline and Inline::C modules be installed on your system. After editing `Comma.pm` to specify SimpleC as the system parser, run `make test` as root. The test scripts should attempt to compile SimpleC and cache the results in Comma's tmp directory. If all goes well, the compiled module will be available to all users of the system. It must be admitted, however, that we have abused the Inline mechanisms a bit to achieve the dynamic loading that Comma's config methods require. If Inline::C passes all its tests, but SimpleC doesn't work for you, don't hesitate to let us know.

Documents and DocumentDefinitions

An XML::Comma system stores pieces of information as *Documents*. The structure and basic behaviors of the Documents in each system are described by *DocumentDefinitions*. This section introduces Documents and DocumentDefinitions. We will mostly refer to Documents as *Docs* and DocumentDefinitions as *Defs*; this saves typing and is consistent with the Perl API.

A Simple Doc and Def

Here is a simple sample Doc, showing the beginnings of a structure that could be used to keep track of information about a registered user of a web site. We'll use this example as we go along, adding features and providing example pieces of code.

```
<User>
  <username>kwindla</username>
  <email>kwindla@xymbollab.com</email>
  <full_name>Kwindla Hultman Kramer</full_name>
</User>
```

That's pretty self−explanatory. The whole thing is XML, with a very simple structure. Here is the corresponding Def:

```
<DocumentDefinition>
  <name>User</name>
  <element><name>username</name></element>
  <element><name>email</name></element>
  <element><name>full_name</name></element>
</DocumentDefinition>
```

Still pretty simple, so far. For your Comma installation to recognize Docs of the *User* type, it suffices to put the above Def in a file called **User.def** somewhere down the **defs_directories** path. If you're following along at the keyboard, you can do that, now, and you'll be able to try out the code examples that follow.

Basic Manipulation: new(), element(), set() and get()

The most basic parts of the Comma API are the methods that manipulate the elements of a Doc. Let's write a little Perl program to make an "empty" User Doc, set its three elements, and then print the result:

```
use XML::Comma;
my $doc = XML::Comma::Doc−>new ( type=>'User' );
$doc−>element('username')−>set ( 'kwindla' );
$doc−>element('email')−>set ( 'kwindla@xymbollab.com' );
$doc−>element('full_name')−>set ( 'Kwindla Hultman Kramer' );
print $doc−>to_string();
```

Running that program should print out something very similar to the sample Doc, above. (The only difference should be that the three elements are not indented. There's a way to do that, too, but we'll cover the subtleties of **to_string()** later.)

What did we do, there? Well, let's take the program line by line.

The first line tells Perl that we're going to be using the XML::Comma framework. All of the Comma modules that we'll need −− such as XML::Comma::Doc −− are pulled in by this statement.

The second line creates a new Doc object. The **Doc−>new()** method takes a parameterized argument **type**, specifying which DocumentDefinition we want our Doc to adhere to.

The next three lines set the contents of the three elements in the Doc. The three statements are completely independent; we could have placed them in any order. We can break these lines up further, to clarify what's going on. Here is the *username* line in two separate statements:

```
my $username_element = $doc−>element('username');
$username_element−>set ( 'khkramer' );
```

First, the **element()** method selects for us the element that we're interested in, taking a single argument −− the name of the element, and returning a reference to an Element object. Then we call that object's **set** method. **set()** takes a single argument, too, a string which will become the content of the Element.

The final line of the little program prints out our Doc. The **to_string()** method generates a string of XML text that completely represents the contents of the Doc.

One more basic method call is worth mentioning here: **get()**. As you might expect, **get()** is the opposite of **set()**. It takes no arguments, and returns the contents of an Element as a string:

```
my $username = $username_element−>get();
```

More Complex Structures: Nested Elements

The Doc so far is very simple: it contains three elements, each of which contain some string−ish content. But we can do better than that, we can introduce elements that, themselves, contain other elements. If we add an *address* element to the Doc, it might look like this:

```
<User>
  <username>kwindla</username>
  <email>kwindla@xymbollab.com</email>
  <full_name>Kwindla Hultman Kramer</full_name>

  <address>
    <street1>922 M Street SE</street1>
    <city>Washington</city>
    <state>DC</state>
    &lt;zip>20003</zip>
  </address>
</User>
```

Corresponding changes in the Def are necessary, of course:

```
<DocumentDefinition>
  <name>User</name>
  <element><name>username</name></element>
  <element><name>email</name></element>
  <element><name>full_name</name></element>

  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
    <element><name>country</name></element>
  </nested_element>
</DocumentDefinition>
```

The new *address* element is declared as a **nested_element**. This means that it will serve as a container for other elements, and will not have content of its own. Comma enforces this distinction between simple and nested elements −− an element can have string content, or it can serve as a container for other elements, but it cannot do both.

You might infer from the above that a nested element will not have **set()** and **get()** methods, but rather, like a Doc, will provide an **element()** method. If so, you infer correctly. To get at the pieces of the address, we can simply "walk down the tree", using the methods we already know about.

```
my $address = $doc−>element('address');
my $formatted_address = $address−>element('street1')−>get() . "\n";
if ( $address−>element('street2')−>get() ) {
  $formatted_address .= $address−>element('street2')−>get() . "\n";
}
$formatted_address .= $address−>element('city') . ',' .
                      $address−>element('state') . '  ' .
                      $address−>element('zip');
```

In fact, a Doc is itself a nested element −− all of the methods that are available for manipulating nested elements are available for Docs, as well. When we talk in more detail about nested elements, we'll often call the nested element a *container*, and the elements that it contains *sub−elements*. Just keep in mind that when we describe nested element operations it doesn't matter whether the container is a Doc or a nested element. In a similar vein, elements can be nested as deeply as you want, you just have to declare the nesting in the Def. (And there's even a way to specify arbitrarily deep recursive nesting, but that's best covered in another section entirely.)

Plural Elements

What if we want to store more than one address. We might, like Amazon, keep a number of shipping addresses on file for each user. To do so, we add a line to the Def, declaring that the address element is *plural*.

```
<DocumentDefinition>
  <name>User</name>
  <element><name>username</name></element>
  <element><name>email</name></element>
  <element><name>full_name</name></element>

  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
    <element><name>country</name></element>
  </nested_element>

  <plural>'address'</plural>
```

```
</DocumentDefinition>
```

Note the quotes around `address`, in the new line. The contents of the **plural** specifier are evaluated as a Perl expression when the Def is loaded into the system, and the return value of that expression must be a list of elements that the system will allow to be *plural*.

We gain a lot of flexibility here, by treating a piece of a Def as a bit of Perl code. The price for this flexibility is a little bit of added complexity: the contents of the **plural** tag must create a valid Perl list. In this case, that means putting quotes around bareword `address`. Many other parts of Comma use this same strategy of embedding Perl code into DocumentDefinitions, and we'll see much more sophisticated examples shortly.

The **element()** method continues to work as it always has. If you re−run the earlier code fragments with the new Def in place, the results will be exactly the same. But our understanding of what **element()** is doing should change a tiny bit: the method doesn't fetch the only matching element for us, it fetches the *first* one. And, because elements don't exist in a Doc until we manipulate them, **element()** must create a new element for us if need be.

For plural elements, we obviously need some more methods. We need a way to fetch elements other than the first one, a way to add a new element, and a way to delete elements that we don't need.

```
# add a new address
my $address2 = $doc−>add_element ( 'address' );
$address2−>element('street1')−>set ( 'PO Box 0000' );
$address2−>element('city')−>set ( 'Anyplace' );
$address2−>element('state')−>set ( 'ZZ' );
# add another new address
my $address3 = $doc−>add_element ( 'address' );
# change my mind, delete that element
$doc−>delete_element ( $address3 );
# get a list of address elements
@addresses = $doc−>elements ( 'address' );
```

The **add_element()** method takes a single argument, the name of the element to add. It creates a new element of the requested kind, appends that element to the container, and returns the newly−created element. To ask a container to add an element that is not plural, if there is already an element of that kind present, is an error. Remember that **element()** auto−creates elements as required, so it is never necessary to call **add_element()** for a non−plural element.

The **delete_element()** method also takes a single argument, but is a bit more complicated. It will accept an element name as string argument, in which case it deletes the last element of that kind. It will also accept an element object, in which case it will delete that specific element. The method returns true if it deletes anything, false if it does not.

The **elements()** method accepts a list of element names and returns a list of the elements of those types, in the order that they exist in the container. (In the above example, we only asked for *address* elements, but we could have asked for *username* and *address* elements, or *username* and *full_name* and *address* elements...)

Actually, the return value of **elements()** is a little trickier than the description above would suggest. In a list context, the method returns an array. But in a scalar context, it returns a reference to an array. This context−awareness makes it possible to write code like:

```
# quick walk down the tree
my $last_street = $doc−>elements('address')−>[−1]−>element('street')−>get();
```

This is usually not a problem; most of the time, things just work out as you would expect them to. If you assign the return value to an array, you get an array. If you dereference with a subscript, you get an element of the list. But there is one very important case that does not work as you would expect. **You can not do the following!!!**

```
# WRONG way to do something if we've got address elements
if ( $doc−>elements('address') ) {...}
```

The above if statement will always be true, because what `if` sees is the reference. Instead, you must use constructions like the following for conditional elements−ing:

```
# do something if we've got address elements
if ( @{$doc−>elements('address')} ) {...}
```

Methods

An **element** holds a piece of information. A **method** generates a piece of information each time it is called. A document definition may supplement its elements, which hold static data, with methods, which return dynamic data.

Suppose we want to provide a method that will display a user's email address modified in such a way as to make things more difficult for the address−collecting web crawlers often used to build spam databases. Here is a method definition that will fetch the contents of the email element, replace the at−sign and periods with text, and return the result:

```
<method>
  <name>email_anti_spammed</name>
  <code>
    <![CDATA[
      sub {
        my $self = shift();
        my $email = $self−>element('email')−>get();
        $email =~ s/\@/ (AT) /;
        $email =~ s/\./ (DOT) /g;
        return $email;
      }
    ]]>
  </code>
</method>
```

A method is expected to have *name* and *code* elements. The *name* is the name by which the method will be called. The *code* element should be text that, when eval'ed, returns a reference to an anonymous subroutine. It is this subroutine that will be called when the method is invoked.

Not too surprisingly, the **method()** routine calls a method. The *email_anti_spammed* method could be used as follows:

```
# set email
$doc−>element('email')−>set ( 'kwindla@allafrica.com' );
# get munged email: kwindla (AT) allafrica (DOT) com
my $munged = $doc−>method('email_anti_spammed');
```

Methods are often most useful at the top level of a document; they function both as bits of reusable code and as programmatic short−cuts. But methods can be defined as "part" of any element −− not just the top−level

Doc. Here is a new definition for the *address* element that includes a method to generate a formatted block of text suitable for printing on an envelope (some of the code inside this method will be familiar from an earlier example):

```
  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
    <element><name>country</name></element>

    <method>
      <name>formatted</name>
      <code>
        <!CDATA[
          # returns a block-formatted address.
          # takes one optional arg indicating whether the country field
          #   should be included: print_country => 1
          sub {
            my ( $self, %args ) = @_;
            my $formatted_address = $self->element('street1')->get() . "\n";
            if ( $self->element('street2')->get() ) {
              $formatted_address .= $self->element('street2')->get() . "\n";
            }
            $formatted_address .= $self->element('city') . ',' .
                                  $self->element('state') . '  ' .
                                  $self->element('zip');
            if ( $args{print_country} ) {
              $formatted_address .= ' ' . $self->element('country');
            }
            return "$formatted_address\n";
          }
        ]]>
      </code>
    </method>
  </nested_element>
```

The *formatted* example demonstrates that methods may make use of arguments. The first argument to **method()** is the name of the method to be invoked; any arguments after that are passed to the invokee. Here is example usage of the *formatted* method:

```
# use a hypothetical &envelope_print sub to generate text on a mailing evelope
envelope_print ( $doc->element('full_name')->get() . "\n" );
envelope_print ( $doc->element('address')->method('formatted', print_country=>1) );
```

Do What I Mean: Shortcut Syntax

The **element()** syntax is quite verbose. Comma provides a more concise syntax that reduces the length and unwieldiness of common method calls. This *shortcut* syntax has a "Do What I Mean" design, which, of course, means that it sometimes doesn't do what you meant.

Shortcuts work via Perl's method `AUTOLOAD` framework. Any Doc or nested element automatically recognizes Perl methods that have the same name as their defined methods and sub−elements. Because our `User` Def defines *username*, *email*, *full_name*, *address* and *email_anti_spammed* elements and methods, all of the following Perl method calls are allowed:

```
# top-level User 'shortcut' methods
$doc->username();
$doc->email();
$doc->full_name();
$doc->address();
$doc->email_anti_spammed();
```

What a shortcut call does depends on what the underlying object referenced is. In the simplest, most useful, and most common case −− here represented by *username*, *email* and *full_name* −− the shortcut fetches the content from the element with the same name as the shortcut.

```
# get username with less typing
my $username = $doc->username();
# which is the same thing as:
my $username = $doc->element('username')->get();
```

If the shortcut is called with an argument, then a **set()** is performed rather than a **get()**.

```
# set the username
$doc->username ( 'kwindla' );
```

In the case of a nested element such as *address*, on the other hand, a **get()** would make no sense. In the case of a singular, nested element, the shortcut call returns the element. In the case of *address* element, which is both nested and plural, the shortcut call returns a list or reference to a list of the *address* elements.

```
# 'address' shortcut
my $first_address = $doc->address()->[0];
# which is the same thing as:
my $first_address = $doc->element('address')->[0];
```

For a Comma method, such as *email_anti_spammed*, the shortcut calls the method. So **$doc−>email_anti_spammed()** becomes **$doc−>method('email_anti_spammed')**. It is possible for a method and an element to have the same name; in this case, the shortcut calls the method rather than accessing the element. **Comma methods *shadow* elements of the same name in the context of shortcut calls.**

A table of shortcuts and their not−short equivalents is probably the easiest way to describe all of the seven possible ways a shortcut can be resolved. Here then, are the many faces of **$x−>foo ([@args])**.

`$x->method('foo', @args)`	If there is a method named *foo*
`$x->element('foo')`	For singular, nested *foo*
`$x->elements('foo')`	For plural, nested *foo*
`$x->element('foo')->get()`	For singular, non−nested *foo* called with no arguments
`$x->elements('foo')->set ( $args[0] )`	For singular, non−nested *foo* called with arguments
`$x->elements_group_get('foo')`	For plural, nested *foo* called with no arguments
`$x->elements_group_add('foo', @args)`	For plural, nested *foo* called with arguments

We've used examples from the top level of the `User` Doc, but short−cut methods are applicable to any nested element context. (Indeed, shortcuts are most useful in terms of keystrokes saved when used to shorten multi−level traversals.) Here is a line of code to grab the zip−code of the first stored address in a `User` Doc:

```
# a shortcut version of $doc−>elements('address')−>[0]−>element('zip')−>get()
$doc−>address()−>[0]−>zip();
```

Nested Element Helper Methods: elements_group_get() and Friends

Shortcuts are one kind of convenience method; they're not strictly necessary but do save typing and make code easier to read. Another set of convenience methods are supported by nested elements: the *group helpers*. These methods make it possible to manipulate instances of a non−nested, plural element as a single group. To demonstrate, we first need to add a simple, plural element to our `User` Def. In an even more contrived attempt to come up with an example than normal, let's allow a user to be known by a number of *nicknames*.

```
<element><name>nickname</name></element>
<plural>'nickname'</plural>
```

A Doc that includes several nickname elements might look like this:

```
<User>
<username>kwindla</username>
<email>kwindla@xymbollab.com</email>
<full_name>Kwindla Hultman Kramer</full_name>
<nickname>Junior</nickname>
<nickname>khkramer</nickname>
<nickname>smooth_operator</nickname>
</User>
```

To add one or more new nickname elements to this Doc, we can use one of the *group helper* methods: **elements_group_add()**. The first argument to **elements_group_add()** is the *name* of the element(s) we'll be adding; the remaining arguments specify the *content* for each new element.

```
# add two nicknames
$doc−>elements_group_add ( 'nickname', 'Sneezy', 'Forgetful' );

# note: the above statement is equivalent to the following two lines of code:
$doc−>add_element('nickname')−>set ( 'Sneezy' );
$doc−>add_element('nickname')−>set ( 'Forgetful' );
```

The opposite function, deleting particular elements from a group, is handled by the **elements_group_delete()** method. Again, the first argument supplies a *name* and the remainder of the arguments specify content strings. If the content of an element matches one of the supplied strings, that element will be deleted. (Any strings that are not matched will be ignored.) If **elements_group_delete** is only given the first, *name*, argument, then all elements in the group are deleted. This provides a convenient idiom for clearing and re−setting an elements group.

```
# remove the nicknames we just added (wrong movie)
$doc−>elements_group_delete ( 'nickname', 'Sneezy', 'Forgetful' );

# remove all the nicknames and replace them with a list of nicknames we
# get back from a couple of subroutine calls
my @new_nicknames = Television::Stooges::nicknames();
push @new_nicknames, Usenet::Rec::Humor::Stooges::FanFiction::nicknames();
$doc−>elements_group_delete ( 'nickname' );
$doc−>elements_group_add_uniq ( 'nickname', @new_nicknames );
```

To query a group for the presence of a particular piece of content, use the **elements_group_lists()** method. This method expects two arguments: *name* and *content*.

```
# check that we really removed the Snow White stuff
print "no more dwarves"
  if  ! $doc−>elements_group_lists('nickname', 'Sneezy')  and
      ! $doc−>elements_group_lists('nickname', 'Forgetful');
```

To slurp the contents of the group's elements into a list, use **elements_group_get()**. As is the case with most of the nested element "plural" methods, **elements_group_get()** returns either an array in a list context and an array reference in a scalar context.

```
# get all of the nicknames
my @nicknames = $doc−>elements_group_get ( 'nicknames' );
# get the last nickname
my $last_nickname = $doc−>elements_group_get('nicknames')−>[−1];
```

Finally, **elements_group_add_uniq()** works like **elements_group_add()** except that it ignores duplicates. If we always use **elements_group_add_uniq()** to add to the nicknames list we will never list a nickname twice.

```
# add a new nickname
$doc−>elements_group_add_uniq ( 'Bashful' );
# add several more nicknames, skipping 'Bashful' because it's already present
$doc−>elements_group_add_uniq ( 'Dopey', 'Bashful', 'Doc' );
```

Whitespace: Ignored and Trimmed

XML−based systems must define how they treat whitespace. HTML, for example, treats all occurrences of whitespace as equivalent. With the exception of content inside a `pre` tag, which is preserved as formatted, there is no difference between a single space and a boatload of carriage returns. (With the exception, of course, of `pre` tags, which preserve whitespace exactly as supplied.)

Comma treats whitespace surrounding its tags as non−meaningful, stripping it all out. The following Docs are exactly the same:

```
<!−− Two equivalent Docs −−>

<User>
<username> kwindla </username>
<full_name> Kwindla Hultman Kramer </full_name>
</User>

<User><username>kwindla</username><full_name>Kwindla Hultman Kramer</full_name></User>
```

Comma's stripping of tag−adjacent whitespace has a very important corrolary: **whitespace is trimmed from the beginning and end of all element content**. So the two **set()** statements below are equivalent, and the string comparison will always be false:

```
# set the username
$doc−>element('username')−>set ( 'kwindla' );
# set the username to the same thing −− whitespace is "trimmed"
$doc−>element('username')−>set ( ' kwindla ' );

# because the whitespace is gone, this can *never* be true
my $matched  =  $doc−>element('username')−>get  eq  ' kwindla ';
```

Of course, the auto−trimming only applies to tags defined in Comma document definitions. It is often convenient to embed XML−marked−up text in a Comma element as "flat" content −− an element that stores an HTML snippet, for example, will include XML tags that have no "meaning" to Comma. Element content is always preserved verbatim (after whitespace is trimmed from the very beginning and very end) by the system; any XML−like strings inside element content are treated exactly like all other text.

XML Escape/Unescape

Every Comma Doc is a syntactically−legal XML document. All tags must be properly balanced and nested, and bare ampersands, left brackets and right brackets must be properly escaped. Elements that contain XML−like tags or markup characters as part of their content will need to take special action to ensure that proper formatting, escaping or CDATA wrapping happens.

Let's add a *bio* element to our `User` Def, and discuss some of the issues involved in storing HTML as element content.

```
<!−− new 'bio' element: holds a chunk of HTML text −−>
<element><name>bio</name></element>

<User>
<username> kwindla </username>
<full_name> Kwindla Hultman Kramer </full_name>
</User>

<bio> Kwin is a programmer who likes <a href="http://use.perl.org">Perl</a>
and <a href="http://www.motorola.com/mcu">6812</a>
assembly language. </bio>
```

The above Doc is perfectly fine. Because the two *a* tags are balanced, the parser has no problem reading in the Doc. After parsing is finished the content of the *bio* element is treated just like any other "flat" piece of content.

We will run into problems, however, if we're not extremely careful about the HTML we try to store in the *bio* element. For example, HTML includes a number of "empty" tags that are usually used in a non−balanced fashion −− *img* and *br*, for example. Unless we force the use of XHTML syntax, which mandates XML−compatible tag usage, we'll need to either escape all mark−up characters or wrap content in a CDATA section.

The utility methods **XML_basic_escape** and **XML_basic_unescape** handle simple escaping and unescaping of markup characters.

```
use Comma::Util qw ( XML_basic_escape XML_basic_unescape );
# escape a string
$escaped = XML_basic_escape ( '<img src="picture.png">' );
$unescaped = XML_basic_unescape ( $escaped );
```

The **set()** and **get()** methods provide a means to escape and unescape strings during get and set operations. If **set()** is called with additional arguments following the *content* arg, they are interpreted as paremeters that effect how the set is performed. The argument **escape=>1** forces the content string to be escaped before other pieces of the set routine −− validation, etc. −− go to work. Similarly, calling **get()** with the parameterized arg **unescape=>1** unescapes the content string before it is returned.

```
# safe set()
```

```
$doc−>element('bio')−>set ( $html_stuff, escape=>1 );

# get() bio content in a string that we can incorporate directly into
# a web page
$doc−>element('bio')−>get ( unescape=>1 );
```

Our other option, as mentioned above, is to "wrap" the bio element's content in an XML CDATA section. The CDATA envelope forces an XML parser to treat the characters inside it as plain text. Comma allows an element to be flagged as CDATA−fied, meaning that on output the entire contents will be wrapped in a CDATA section. Comma treats this CDATA facility as high−impact and coarse−grained. As a result the declaration is a one−way street: once a CDATA element, always a CDATA element. The **wrap_cdata()** method flips the switch, so to speak.

```
# configure the bio element so that it always CDATA−wraps its content
$doc−>element('bio')−>wrap_cdata();
# now we can set() with impunity
$doc−>set ( $messy_html );
```

The **to_string()** method on the CDATA−set element will produce output that looks something like this:

```
<bio><![CDATA[Kwin is a programmer who likes <a href="http://use.perl.org">Perl</a>
and <a href="http://www.motorola.com/mcu">6812</a>
assembly language.]]></bio>
```

Automatic Content: <default>

It is often useful to define *default* content for a class of elements, content that **get()** will return for any instance of an element that doesn't have content of its own. We can amend the definition of the *bio* element (defined in the previous section) to provide a standard "no information available" string if a `User` Doc doesn't include a bio.

```
<element>
  <name>bio</name>
  <default>No bio information available.</default>
</element>

# set() bio information
$doc−>element('bio')−>set ( 'Kwindla is a programmer' );

# get() will return our new bio −− this prints out 'Kwindla is a programmer';
print $doc−>element('bio')−>get();

# "clear" bio content by passing set() an undef argument
$doc−>element('bio')−>set();

# now get() will return our default string −− 'No bio information available'
print $doc−>element('bio')−>get();
```

As the above code demonstrates, calling **set()** with an undefined value as its content argument (which passing no arguments does implicitly) "clears" the content of an element, and any subsequent **get()** calls will again return the default string. Note that only an `undef` argument will clear an element's content; in particular, an empty string is perfectly valid as content and a **get()** on an element with an empty string as its content will happily return that empty string.

It is sometimes important to differentiate between an element that doesn't have any content and an element that has the same content as its Def's default string. The **get_without_default()** method returns an element's content exactly as is, without falling back to any default value that may be defined. Unlike **get()**, which returns an empty string if there is neither element content nor Def default, **get_without_default()** returns `undef` if an element has no content at all.

Storing Dynamic Information in Defs: pnotes

Document definitions are static constructs. However it can be useful to tie some dynamic bits of information −− status or state flags, simple lookup tables and the like −− to a def.

To enable a Def to "hold" some long−lived bits of dynamic information, each def exposes a unique **pnotes** hash, available to any piece of code in the system. (Comma borrowed the idea for, and the name of, the **pnotes** hash from Apache.)

```
# a bit of pnotes manipulation

my $def = XML::Comma::Def−>read ( name=>'some_docdef' );
$def−>def_pnotes()−>{'foo'} = 'bar';

# prints out 'Foo from def: bar'
print "Foo from def: " . $def−>def_pnotes()−>{'foo'} . "\n";

my $doc = XML::Comma::Doc−>new ( type => 'some_docdef' );

# prints out 'Foo from doc: bar'
print "Foo from doc: " . $doc−>def_pnotes()−>{'foo'} . "\n";

# prints out 'Foo from pathname: bar'
print "Foo from pathname: " . XML::Comma−>pnotes('some_docdef')−>{'foo'} . "\n";

# and every element down a def's tree has its own pnotes, too
XML::Comma−>pnotes('some_docdef:nested_element:another_element')−>{'test'} = 'Ok';
print "Ok down longer pathname: " . XML::Comma−>pnotes('some_docdef:nested_element:another_elemen
```

There are three new methods here. Each element exposes a **def_pnotes()** method, which returns a reference to that element's def's pnotes hash. Each def also exposes a **def_pnotes()** method, which returns a reference to its own pnotes hash. The two methods are "different but the same" −− for convenience, you can call **def_pnotes()** on an element or on that element's def and get back the same hash reference.

The third new method is the system call **XML::Comma−>pnotes()**, which takes a pathname and returns that def path's pnotes hash.

Storage and Retrieval

Manipulating Docs in memory is only a small part of the story. We need a way to store Docs in permanent collections, a way to retrieve these permanently stored Docs, and a way to manipulate the collections themselves.

The Store Definition

Let's introduce a new section to the *User* Def: **store**.

```
<DocumentDefinition>
  <name>User</name>
  <element><name>username</name></element>
  <element><name>email</name></element>
  <element><name>full_name</name></element>

  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
  </nested_element>

  <plural>'address'</plural>

  <store>
    <name>main</name>
    <base>comma_guide</base>
    <location>Sequential_file</location>
  </store>
</DocumentDefinition>
```

This is the simplest possible store specification: we supply a **name**, a **base** directory and a **location**.

The **name** element specifies how we'll refer to this particular store. As with elements, we can specify more than one store, so we need names to differentiate one from 'nother. We've called this particular store *main*.

The **base** element supplies a directory, underneath the document root, where we're going to put the Docs that we're storing. For this store, since the base is `comma_guide`, all of the storage will take place in `<document_root>/comma_guide/`.

The **location** element specifies how Docs will be stored within the **base** context. In this case we're storing Docs in a series of sequentially−numbered files.

Two Methods: store() and retrieve()

With this definition of our *main* store in place, we're ready to store and retrieve User documents.

```
# make a new Doc, so we have something to store.
my $doc = XML::Comma::Doc−>new ( type=>'User' );
$doc−>element('username')−>set ( 'kwindla' );
$doc−>element('email')−>set ( 'kwindla@xymbollab.com' );
```

```
# write this Doc out to the "main" permanent store
my $key = $doc−>store ( store => 'main' )−>doc_key();
# now read the Doc back in, manipulate it, and store it back out to the same place
my $d2 = XML::Comma::Doc−>retrieve ( $key );
$d2−>element('full_name')−>set ( 'Kwindla Hultman Kramer' );
$d2−>store();
```

There are three new methods here −− **store()**, **retrieve()**, and **doc_key**.

The **store()** method writes a Doc out to permanent storage. A **store => <name>** argument must be supplied the first time the method is called on a new Doc, to specify which of the stores in the Def will be used. The **store()** method re−returns a reference to the Doc, so that you can chain method calls together easily. The **doc_key** method returns a unique, long−term identifier for the stored Doc.

The **retrieve()** method fetches a Doc out of storage, and expects to be supplied a document **key** as its argument.

Where Are the Files?

It's worth looking at the files that **store()** writes out. If you ran the above bit of code, you should be able to look in your document root and see a directory named `comma_guide`. In that directory, there should be a file named `0001`. (And if you ran the code multiple times, also `0002` `0003`, etc.) The contents of these files should look familiar: the text in them was produced by an internal call to **to_string()**. We can compare the output from a **to_string()** call with the contents of a store file, to confirm this:

```
my $store = XML::Comma::Def−>read(name=>'User')−>get_store('main');
my $doc = XML::Comma::Doc−>retrieve ( type => 'User',
                                      store => 'main',
                                      id => $store−>first_id() );
# print out the doc with a to_string()
print "doc retrieved...\n"
print "  key: " . $doc−>doc_key() . "\n";
print "  from to_string()...\n";
print "−−−−\n";
print $doc−>to_string();
print "−−−−\n";
# cat the file that we got the doc from
print "  from file: " . $doc−>doc_location() . "\n";
open ( FILE, '<'.$doc−>doc_location() );
my @lines = <FILE>;
close ( FILE );
print "−−−−\n";
print @lines;
print "−−−−\n";
```

We've snuck several things into the above example.

In the first line we **read()** the `User` Def. This is the Def that we've been adding to as we go along in this chapter, but here we're going to be querying it programmatically, rather than editing it as a text file. **Def−>read()** gives us a reference to the Def object, upon which we immediately call **get_store()** to get a reference to our `main` store. We use that to get the **id** of the first document we stored in `main`, whatever and whenever that was. A document **id**, as you might guess, is one of the parts that makes up a document key. (The other mandatory parts are a document type and a store name.) As you can see, **retrieve()** is flexible: it accepts a single argument and interprets that as a key (as in the previous example); it is also happy to accept separate, parameterized arguments supplying a type, store name and Doc id, which is what we've done here.

Again, we see the **doc_key()** method, which returns this Doc's key, and a new method, **doc_location()**, which returns the underlying file that this Doc was fetched from. It is worth noting that **doc_location()** is rarely used in the course of "normal" Doc manipulation, because Comma handles all of the underlying filesystem tasks that are part of ordinary storage, retrieval and the like.

There are other "doc_foo()" methods, including **doc_store()** which returns a reference to the store that was used to fetch or store the Doc, and **doc_id()**, which returns the Doc's id. It is an error to call any of the doc_foo() methods on a newly−created Doc that has not yet been stored.

Multiple Users and Processes: Permissions and Locking

Access permissions are an important part of any multi−use system. XML::Comma uses the underlying filesystem to provide basic permissions facilities. The store definition may include a **file_permissions** element, which sets the rwx permissions on any stored files. Here is our `main` store with a new line that makes these files world−readable but writable only by their owner:

```
<store>
  <name>main</name>
  <base>comma_guide</base>
  <location>Sequential_file</location>
  <file_permissions>644</file_permissions>
</store>
```

The `644` specification is suitable for a system in which all `User` editing is done by processes running as a single user, but in which many users might need to run processes that need read−access to `User` information. It is actually more common for a group of users to need write access to a Doc collection; for that reason the default value of the **file_permissions** element −− the value that is used by the system if no specifier is given −− is **664**.

Because Comma depends on the filesystem to manage permissions, you will need to understand how the filesystem determines and applies permissions information to/for individual files in order to set up complicated scenarios. Remember that Comma code always runs as part of some particular process, under the ownership of a specific user.

Permissions restrictions address issues of information ownership and security. File permissions discriminate among multiple users of a system. An even more fundamental set of problems is posed by the multi−process nature of the systems on which Comma runs. We must be able to **lock** Docs so that concurrent processes do not simultaneously attempt to modify a file.

The **retrieve()** method automatically acquires a **lock** on the requested Doc. As long as this lock is held, the Doc cannot be retrieved again. The **store()** method automatically unlocks the stored Doc.

Because of the automatic locking, **retrieve()** is a relatively heavy−weight method. In addition, if **retrieve()** cannot immediately acquire its lock, it waits −− re−trying periodically −− until it finally can. The **retrieve()** method should therefore be used carefully, with the time that a Doc is held open kept as short as possible. (An optional argument to retrieve, **timeout=><seconds>** is also available. With a timeout specified, **retrieve()** will throw an error if it is unable to acquire its lock within the given number of seconds.)

The **read()** method is an alternative to **retrieve()**, for situations in which a Doc will be read but not modified. In fact, in most applications, **read()** is by far the most common access method. Because **read()** does not need to acquire a lock, it is somewhat faster than **retrieve()**. The two methods take the same arguments.

There is one other method in the retrieve family: **retrieve_no_wait()**. This method is exactly like **retrieve()**, except that if it fails to immediately acquire a lock it returns `undef`, rather than blocking. Programmers with extensive experience designing multi−threaded/concurrent systems will find uses for this method: other programmers will find abuses. In general, if you can't describe in exact and minute detail why you are using **retrieve_no_wait()**, you shouldn't be.

As the necessary complement to **retrieve()**, **store()** must unlock objects as they are written out to permanent storage so that other users of the system will be able to fetch them. After storage, a Doc object becomes read−only, as if it had been opened with **read()**.

It is possible to **store()** a Doc without unlocking it (useful, for example, to write out intermediate changes as part of a series of operations). The **keep_open=<true>** argument specifies that the lock be retained. (Conversely, a Doc that has been opened read−only can be locked with the **get_lock()** or **get_lock_no_wait()** methods.)

Finally, the methods **erase()**, **copy()** and **move()** perform the operations that their names suggest:

```
# retrieve and then erase a Doc
my $doc = XML::Comma::Doc−>retrieve ( $key_a );
$doc−>erase();
# retrieve and the move a Doc
$doc = XML::Comma::Doc−>retrieve ( $key_b );
$doc−>move ( store=>'other_store' );
# read and copy a Doc (we're not modifying the original, so it's
# okay to read() instead of retrieve()
$doc = XML::Comma::Doc−>read ( $key_c );
$doc−>copy ( store=>'other_store' );
```

As a side note, **copy()** and **move()** accept the same arguments as **store()**, including **keep_open=<true>**, and you should always supply a **store=><name>** when copying and moving −− the normal use of these methods is to transfer a Doc from one store to another. (Confusingly, in this normal case, **copy()** is really just a synonym for **store()**; calling **store()** with a *new* **store=><name>** specifier effectively performs a copy. The only case in which the actual **copy()** method is uniquely required is the copying of a Doc *within* the same store.)

Iterating Over Stored Docs

It is often necessary to process some or all of the Docs in a store. Methods exist to fetch the first and last ids in a store and, given an id, to fetch the ids before and after it. In one of the examples above we retrieved the first Doc in the `main` store. We'll begin with that same code, and then go on to iterate through all of the Docs in the store.

```
my $store = XML::Comma::Def−>read(name=>'User')−>get_store('main');
my $doc = XML::Comma::Doc−>retrieve ( type => 'User',
                                      store => 'main',
                                      id => $store−>first_id() );
print "first doc: " . $doc−>doc_key() . "\n";
while ( my $id = $store−>next_id($doc−>doc_id()) ) {
  $doc = XML::Comma::Doc−>retrieve ( type => 'User',
                                     store => 'main',
                                     id => $id );
  print "next  doc: " . $doc−>doc_key() . "\n";
}
```

This code uses the store's **first_id()** and **next_id()** methods. To iterate in the other direction, we could substitute **last_id()** and **previous_id()**.

The **prev_** and **next_** methods are fine for fetching a few docs, but for sizable loops they are a little clumsy and a lot slow. An *iterator* provides a means by which to apply repetitive operations to a set of stored documents quickly and easily.

```
# basic iterator -- start from the end and work backwards
my $iterator = $store->iterator();
while ( my $doc = $iterator->prev_read() ) {
  print "working on doc: " . $doc->doc_id() . "\n";
}

# with some additional parameters -- start from the beginning and
# limit the set to the first 500 docs
$iterator = $store->iterator ( size=>500, pos=>'-' );
while ( my $doc = $iterator->next_read() ) {
  print "working on doc: " . $doc->doc_id() . "\n";
}
```

An iterator is obtained by calling the store's **iterator()** method. By default, **iterator()** provides access to all of the store's documents, starting with the last doc. (This is the default because iterating backwards over recently−stored docs is a fairly common thing to do.) Two arguments to **iterator()** modify this default behavior: **store=>** limits the size of the iterator's result set, and **pos=>** specifies whether the iterator is initially set to point at the end or at the beginning of the set −− **'+'** specifies the end (and is equivalent to the default of not specifying a pos), and **'−'** specifies the beginning. The **size=>** argument can only be used to pick out the first or last *n* documents. There is no way to pull a subset of documents out of the "middle" of a store. When used with **pos=>'−'**, the size specifier will select documents from the beginning of the store, and when a **pos=>** argument is not given (or when **pos=>'+'** is specified), the size specifier will select documents from the end of the store.

The basic iterator methods are **next_id()**, **prev_id()**, **next_read()**, **prev_read()**, **next_retrieve()**, and **prev_retrieve()**. The names are pretty self−explanatory. Each of these methods returns an id or doc, as the case may be, unless the iterator has passed the beginning or end of its collection, in which case the method returns `undef`. The six methods can be called in any combination and in any order. (Criticism−inclined readers may, at this point, be thinking that "iterator" is a poor name for this class, given that it is possible to move across the set in any order and backwards and forwards. Those readers are probably correct.)

Four more methods are defined for advanced mucking around with an iterator. These methods should be wielded with caution, as they are not usually needed and they don't do any error or sanity checking. The **length()** method returns the size of the iterator's document set; the **index()** method gives the position of the current pointer into that document set; the **inc()** method moves the pointer a relative amount −− with no argument **inc()** adds one to the pointer, given an argument it adds that value to the pointer (`−1` is a common argument); and the **set()** method sets the pointer to an absolute index value −− so `$iterator−>set($iterator−>length())` would reset an iterator such that the next call to **prev_id()** will fetch the last id in the set.

Location Chains

So far, our storage definition for `main` has used only a single **location** element. We saw above that specifying `Sequential_file` governed the "file" portion of the storage location. To understand how to create more complex storage patterns, we need to understand how multiple location specifiers can be "chained" together.

A filesystem is a heirarchical store: directories contain files and directories, which contain more files and directories, which contain more files and directories, ad infinitum. Each time a Doc is stored, Comma uses the **document_root**, the **base** specifier and the **location** elements in a storage definition to build a "location chain" that determines where in the filesystem to save the written−out Doc. For our `main` store, the chain looks like this:

document root	base	location
`XML::Comma−>document_root()`	`comma_config`	`<location>Sequential_file</location>`

There are other location specifiers besides `Sequential_file`. Some of these are designed to be used in pairs or groups, so that several location specifiers can be combined as part of a chain. One of these "intermediate" specifiers is `Sequential_dir`, which is similar to `Sequential_file` except that it determines an intermediate directory in the location chain rather than a final file. Here is our store definition with a new addition:

```
  <store>
    <name>main</name>
    <base>comma_guide</base>
    <location>Sequential_dir</location>
    <location>Sequential_file</location>
  </store>
```

The first file stored by this store will be located at:

```
<document_root>/comma_guide/0001/0001
```

We've added a directory level to the chain; the first `0001` comes from the `Sequential_dir`, the second from the `Sequential_file`. One effect of this addition is to increase the capacity of the store. We're limited to 9999 files per directory, so before we could store a maximum of 9999 Docs and now we can store up to 9999 * 9999, or 99,980,001. And we can add as many `Sequential_dir`s to the chain as we like, increasing the number of directories in the resulting storage locations.

Location specifiers often accept arguments that further determine how they behave in the chain. `Sequential_file` recognized two arguments, and `Sequential_dir` recognizes one. Here is another modified version of our storage definition:

```
  <store>
    <name>main</name>
    <base>comma_guide</base>
    <location>Sequential_dir:max,10</location>
    <location>Sequential_file:max,99,extension,'.xml'</location>
  </store>
```

Now each of the location specifiers has an arguments list attached. A colon separates the specifier name from the arguments, and the arguments themselves take the form of a Perl list, which will be turned into a hash of key/value pairs when the definition is loaded.

The first argument is common to both declarations: **max** specifies the maximum number of files that will be allowed in this part of the chain. (When we stated above that we were limited to 9999 files, we were referring to the default value of the **max** argument. If we had wanted to square the capacity of the storage without adding an intermediate directory, we could have simply specified **max,99_980_001** as an argument to `Sequential_file`. Doing so has a serious drawback, however; finding, creating and deleting files gets progressively slower as the number of files in a directory climbs.)

`Sequential_file`s second argument, **extension**, provides an extension to be tacked onto the end of every Doc's storage file. This can be useful if, for example, other tools for managing or manipulating files will co−exist with XML::Comma in a given application. With our most recent definition, the first and last files in the a store would have the following locations:

```
<document_root>/comma_guide/01/01.xml
<document_root>/comma_guide/10/99.xml
```

The Storage in More Detail section provides additional information about storage definitions, including documentation for all of the standard location modules.

Validation, Macros and Hooks

Document Definitions describe and constrain the basic structure of the documents that we can produce. For example, an attempt to make use of an element that isn't specified in a document's Def generates an error. This section describes Comma's mechanisms for "validating" the structure of documents and the content of elements.

Document Structure: Required Elements and validate()

Section Three introduced the **plural** tag. This tag determines which elements may exist multiple times in the given container. Another container−level tag is **required**, which specifies that a container must include at least one of each of the specified elements. Here is our `User` Def with a new validity constraint:

```
<DocumentDefinition>
  <name>User</name>
  <element><name>username</name></element>
  <element><name>email</name></element>
  <element><name>full_name</name></element>

  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
  </nested_element>

  <plural>'address'</plural>
  <required>qw( username email full_name )</required>

  <store>
    <name>main</name>
    <base>comma_guide</base>
    <location>Sequential_file</location>
  </store>
</DocumentDefinition>
```

To be "valid," a `User` Doc must now have content in its *username*, *email* and *full_name* elements. A document that is not valid cannot be stored −− the storage routines all call the method **validate()**, which throws an error if all required elements are not present. The **validate()** method can also be called directly. It takes no arguments and returns the emtpy string; it's only function is to throw an error if the Doc doesn't pass all validity tests. Here are two simple code snippets, for more information see the section on errors and error handling:

```
# check whether a Doc passes validity tests
eval {
  $doc−>validate();
}; if ( $@ ) {
  print "doc didn't validate: $@\n";
}

# the same idea, but during a store()
my $key
eval {
  $key = $doc−>store( store=>'main' );
}; if ( $@ ) {
  print "doc couldn't be stored: $@\n";
} else {
  print "doc was stored successfully: $key\n";
}
```

Our example use of **required** is not very complicated. As with all things to do with nested elements, **required** and **validate()** are just as applicable deep inside a nested structure as at the very top level. Any nested element can specify a **required** list, and can be checked with a call to **validate()**. More interestingly, calls to **validate()** automatically check the validity of all elements underneath the caller, so a Doc−level validity check walks the entire document tree. This is convenient and it makes good theoretical sense: no element can be valid that itself contains invalid parts.

Element Content: Macros and validate_content()

A container's validity is a function of the sub−elements that it contains. A simple element's validity is a function of its contents. A *macro* defines and limits the type of content that an element may have. Here is our `User` Def with macros added to its *username* and *email* definitions.

```
<DocumentDefinition>
  <name>User</name>

  <element>
    <name>username</name>
    <macro>length:min,4,max,20</macro>
  </element>

  <element>
    <name>email</name>
    <macro>email</macro>
  </element>

  <element><name>full_name</name></element>

  <nested_element>
    <name>address</name>
    <element><name>street1</name></element>
    <element><name>street2</name></element>
    <element><name>city</name></element>
    <element><name>state</name></element>
    <element><name>zip</name></element>
  </nested_element>

  <plural>'address'</plural>
  <required>qw( username email full_name )</required>

  <store>
    <name>main</name>
    <base>comma_guide</base>
    <location>Sequential_file</location>
  </store>
</DocumentDefinition>
```

We can use the **validate_content()** method to check whether a string can be accepted as an element's content. The method takes a single argument −− the prospective content −− and throws an error if the content fails to pass the validity checks. It is not usually necessary to call **validate_content()** directly, because **set()** calls the method at the very beginning of its operation, before doing anything else. Here is a typical bit of error−checked **set()** code:

```
# modifying a User doc
eval {
  $doc−>username ( $username );
  $doc−>email ( $email );
  $doc−>full_name ( $full_name );
}; if ( $@ ) {
  handle_content_error ( $@ );
}
```

The **validate()** method is also defined for non−nested elements. It is possible to use the unsafe **append()** method to construct invalid element content (and also possible to read invalid Docs out of storage). The **validate()** method checks an element's existing content for validity. Just as with nested elements, this method is called by all of the storage methods, so that an invalid document will not be written out to permanent storage.

As with storage location specifiers, the **macro** tag should contain a name followed by an optional argument list (with a colon in between). Different macros expect different numbers of arguments and different argument formats. The *enum* macro, for example, takes a list of strings that will be the only acceptable contents for the element being defined. Let's add a new *subscription* element to the `User` Def, indicating what level of service a user has paid for. (This time, we won't re−produce the whole Def, just the new element.)

```
<element>
  <name>subscription</name>
  <macro>enum: 'basic', 'premium', 'lifetime'</macro>
</element>
```

There are only four possible values for the content of the *subscription* element: `undef`, *basic*, *premium*, and *lifetime*. "Hmm, `undef`" you say, "I don't see `undef` in that list? Well, *enum* always includes `undef` as an implicit member of the possible−contents list. The reason for this will be clear after a little reflection: because Comma treats a content−less element as indistinguishable from an element that is not there at all, `undef` must be legal content for all elements. To make an empty element illegal is actually the same operation as to make it required. If we want every `User` Doc to include subscription information, we can define the *subscription* element to be **required**:

```
<element>
  <name>subscription</name>
  <macro>enum: 'basic', 'premium', 'lifetime'</macro>
</element>

<required>'subscription'</required>
```

More Flexibility: Perl Hooks

The **required** and **macro** facilities that we've just seen are actually implemented using a finer−grained, more flexible tool: the *hook*. A hook is a piece of Perl code that will, under specific circumstance, be automatically called by the Comma system. Declaring an element as required actually installs a **validate_hook** −− the **required** tag is just a short−cut, provided because the facility is so important and so commonly used. The following two pieces of a hypothetical Def are exactly equivalent:

```
<!−− 1) a required tag specifying two element names−−>
<required>'foo', 'bar'</required>
```

```
<!-- 2) the two validate_hooks that are actually installed
when the Def is parsed, one for 'foo' and one for 'bar' -->

<validate_hook>
<![CDATA[
  sub {
    my $self = shift();
    my $req_el = \$self->elements('foo')->[0];
    die "required element 'foo' not found in " . $self->tag_up_path() . "\n" if
                    (! $req_el) or
                    ((! $req_el->def()->is_nested()) and (! $req_el->get()));
  }
]]>
</validate_hook>

<validate_hook>
<![CDATA[
  sub {
    my $self = shift();
    my $req_el = \$self->elements('bar')->[0];
    die "required element 'bar' not found in " . $self->tag_up_path() . "\n" if
                    (! $req_el) or
                    ((! $req_el->def()->is_nested()) and (! $req_el->get()));
  }
]]>
</validate_hook>
```

This example demonstrates the common convention for writing hooks: most hooks are subroutines that are compiled into code references when the Def is loaded by the system; they can expect to be passed certain arguments when they are invoked; they should make use of the Comma API to do whatever work they need to do; and they should return appropriate values or throw errors, depending on what is expected of them.

We can go over the first of these hooks line by line. (The second hook is exactly the same, except that `'bar'` is substituted for `'foo'` in two places.) The first line is an opening CDATA tag. Perl snippets usually include characters that are illegal in XML −− the arrow operator is particularly common in this kind of code −− so wrapping the content in a CDATA section is a near necessity. The second line begins an anonymous subroutine declaration. The third line establishes a named variable, `$self`, which comes from the first argument to the sub. The next line fetches the first 'foo' element, if any, into `$req_el` −− the `$req_el` variable now holds either an element object or an undefined value. The final statement throws an error if either `$req_el` is not defined, or `$req_el` is a non−nested, empty element. (NOTE: FIXING the obvious bug here, real real soon.)

The required example demonstrates the use of a **validate_hook** in the "structural" context −− checking the sub−elements of a nested element. We can use the same approach to validate the contents of a non−nested element, but in this case we must expect two arguments: the element and the proposed new content. Imagine, if you will, an element, *<delicate_sensibilities>*, which must contain text that will not shock or offend children, great aunts and members of the clergy. Imagine, also, a hypothetical CPAN module Lingua::FCC_Check, which can check for words that are proscribed by the Federal Communications Commission from over−the−air broadcast in the United States. Here, then, is a definition for the *<delicate_sensibilities>* element:

```
<element>
  <name>delicate_sensibilities</name>
  <validate_hook>
  <![CDATA[
    use Lingua::FCC_Check;
    sub {
      die "unacceptable language detected for " . $_[0]->tag_up_path() . "\n"  if
        Lingua::FCC_Check::check ( $_[1] );
    }
  ]]>
  </validate_hook>
</element>
```

The only new thing in this example is the `use` statement that precedes the subroutine definition. We need to pull in the Lingua::FCC_Check module, so we do that just as we would in a stand−alone program.

To summarize, **validate_hook**s may be defined for both simple and nested elements and should take the form of anonymous subroutines. In the case of a nested element, the hook expects the element itself to be the sole argument. In the case of a non−nested element, the hook expects to be passed the element and a string containing the content to be checked. The **validate()** method calls any hooks that have been defined for an element; as does **validate_content()**. More hooks (called as part of storage, indexing, etc.) will be introduced as we go along, and documentation for all available hooks can be found in the hooks reference section.

Writing New Macros

Macros were designed as a way to extend the syntax of document definitions without modifying any of the Comma system code. When an element definition is loaded, any macros that it contains are given a chance to execute. Writing and installing macros is relatively easy. In general, macros work by installing hooks, so you've already seen most of what you need to know to create a new macro.

For a macro to be available to the system, the definition loader must be able to find it. The loader will look in the same places that it looks for defs (the list of directories in the **defs_directory** Comma variable), and it will look in files named *macro.extension*, where *macro* is the name of the macro and *extension* is the string defined by the **macro_extension** variable.

To turn the "FCC_Check" example from the previous section into a macro, we need to save a few lines of perl code in a file that meets the above criteria (on my system, I'm using /comma/defs/macros/fcc_approved.macro). Here is the contents of the file:

```
# fcc_check: a macro to check element content for blue language

use Lingua::FCC_Check;
$self->add_hook ( 'validate_hook',

  sub {
    die "unacceptable language detected for " . $_[0]->tag_up_path() . "\n"  if
        Lingua::FCC_Check::check ( $_[1] );
  }

);
```

The first line is just a comment that helps us remember what this code snippet does, if we run across it in an unexpected place. The `use` statement and the subroutine definition are familiar from the validation hook version of this code. The only new thing here is the **add_hook()** method. The syntax is a little hard to see, but **add_hook()** is quite simple: it expects a hook name as its first argument, and a subroutine reference or string (which will be eval'ed and must become a subroutine reference) as its second argument. The subroutine is installed as a hook of the requested type.

Turning the FCC check into a macro simplifies the definition of the *delicate_sensibilities* element considerably. Even more imporantly, we can re−use this macro in any other Def on this system, and changes −− bug fixes, new additions to the FCC list −− will only need to be made to the macro, not to each element that defines the hook.

```
# the new, improved delicate_sensibilities def
<element>
  <name>delicate_sensibilities</name>
  <macro>fcc_approved</macro>
</element>
```

The *range* macro (part of the standard Comma installation) provides a slightly more complex example. This macro is used to limit content to a range of numbers, for example between one and ten. As such, *range* requires two arguments; the first specifies the low end of the range and second the high end.

```
# range macro: takes two arguments, low−end and high−end

my $low = $macro_args[0];
my $high = $macro_args[1];

$self−>add_hook ( 'validate_hook',

  sub {
    my ( $doc, $content ) = @_;
    if ( $content < $low  or
         $content > $high ) {
      die "'$content' out of range ($low:$high)\n";
    };
  }

);
```

The only thing here that we haven't seen before is the pre−defined variable **@macro_args**. Like **$self**, the **@macro_args** array is filled with the appropriate values by the macro loader. Macros can do whatever they want with the arguments that are supplied them. This macro simply makes use of the first two elements in the list as part of the hook subroutine. (It should actually probably do a little bit of pro−active error checking.) Here is how we might use the *range* macro.

```
<element>
  <name>one_to_ten</name>
  <macro>range:1,10</macro>
</element>
```

Indexing

XML::Comma implements *storage* and *indexing* separately.

Comma *storage* generally involves writing complete documents out to disk. Each stored document is retrievable by a unique key, and collections of stored documents can be iterated across in key order. Most of the time, stored documents are saved as normal, XML−formatted text files. Modern filesystems are fast, robust and well understood. Relying on the standard filesystem functionality enables a systems administrator to use normal tools for backup, maintenance and monitoring, and allows programmers to use standard utilities for quick or simple manipulations. (It is very convenient, for example, to be able to do a quick `grep` on a directory full of Docs.)

Comma *indexing* involves saving pieces of documents in a relational database so that complex search, sort and retrieval operations can be performed flexibly and efficiently. These tasks are "above and beyond" what a filesystem is capable of, so Comma builds its indexing functionality as a relational database framework. The system can be configured to use any RDBMS; Comma provides a standard interface that sits atop the sophisticated storage and query capabilities of platforms such as MySQL, Postgres or Oracle.

A `User` Index Definition

An index allows a collection of Docs to be searched and sorted according to their elements' contents. An index may hold only one type of Doc. We'll build an index for our `User` Docs to demonstrate the basic features of the indexing framework.

An index defines one or more **fields**, with each field normally corresponding to an element or method in the document definition. A simple index might only contain a single **field**:

```
<DocumentDefinition>
<name>User</name>
<element><name>username</name></element>
<element><name>email</name></element>
<element><name>full_name</name></element>

<nested_element>
<name>address</name>
<element><name>street1</name></element>
<element><name>street2</name></element>
<element><name>city</name></element>
<element><name>state</name></element>
<element><name>zip</name></element>
</nested_element>

<plural>'address'</plural>

<store>
<name>main</name>
<base>comma_guide</base>
<location>Sequential_file</location>
</store>

<index>
<name>main</name>
<field><name>email</name></field>
</index>
```

```
</DocumentDefinition>
```

With the *main* index part of our document definition, we can use the **index_update()** method to add documents to it. Calling **index_update()** −− which takes as its **index=>** argument the name of the index to update −− adds a document to an index or, if the document is already present, updates the index to reflect any changes.

```
# add/update this Doc's record in the 'main' index
$doc−>index_update ( index => 'main' );
```

On the other hand, **index_remove()** deletes a document from an index. Like **index_update** it expects the name of an index as an **index=>** argument.

```
# delete this Doc's record in the 'main' index
$doc−>index_remove ( index=>'main' );
```

Querying the Index: Iterators

We need a way to get at the `User` Docs that are in our index. First we need a handle to the index itself. Then we can ask the index for an **iterator** that will step through all of the Docs:

```
# get 'main' index
my $index = XML::Comma::Def−>read(name=>'User')−>get_index ('main');
# get iterator
my $i = $index−>iterator();
# iterate, printing out "$key: $email"
while ( $i ) {
  print $i−>doc_key() . ': ' . $i−>email() . "\n";
  $i++;
}
```

There are several new methods here. The **get_index()** method operates like **get_storage()**, taking a single argument and returning the index of that name. The index's **iterator()** method returns an iterator object, which provides a means to step through the documents in the index. An iterator can only deal with documents one at a time, and can only advance in one direction through its sequence. Here, we use the ++ operator to advance the iterator.

Every iterator exposes its fields as methods, so we call the **email()** method to get the value of this record's *email* field −− a value which came originally from the *email* element of the Doc that this record represents.

Every iterator implicitly includes the **doc_key** and the **doc_id** as fields, so **doc_key()** and **doc_id()** are always available as methods. Another special method, **record_last_modified()** is also available. As its name suggests, **record_last_modified()** returns a timestamp (unix system time) indicating when the index record was last changed.

The ++ operator is actually a short cut for a named method, **iterator_next()**. And if that isn't bad enough, there's an implicit check of the **iterator_has_stuff()** method triggered by the boolean context of the `while` statement. The implicit−nesses are an example of operator overloading, which is exmplained in detail in the Camel book. For our purposes, suffice it to say that 1) it is easy and correct to write an iterator loop as above, and 2) you've just seen the only two overloaded operators that the `Iterator` class defines −− ++ => **iterator_next()** and **boolean−ization** => **iterator_has_stuff()**.

In general, I prefer compact idioms, and the simple ++ loop is both compact and (to me, anyway) highly readable. However, in the spirit of over−explanation, here are several exactly−equivalent versions of the lowly iterator loop. (And a note to careful observers: it doesn't matter that some of these loops "increment" the iterator on loop entry and some don't −− an iterator contrives to point to its first record in a "lazy" fashion so that programmers don't ever have to worry about whether an iterator is newly−created or not.)

```
my $i = $index−>iterator();
while ( $i++ ) {
  $i−>foo();
}

my $i = $index−>iterator();
while ( $i−>iterator_has_stuff() ) {
  $i−>foo();
  $i−>iterator_next();
}

my $i = $index−>iterator();
while ( $i ) {
  $i−>foo();
  $i−>iterator_next();
}

my $i = $index−>iterator();
while ( $i−>iterator_next() ) {
  $i−>foo();
}

my $i = $index−>iterator();
while ( $i ) {
  $i−>foo();
  $i++;
}
```

NOTE −− there is a bug in perl 5.6.x and 5.8.0 that makes the first (and most compact) idiom above leak memory. The iterator object won't get properly garbage collected, when the `while()` look is written like that. This can be a big problem in long−running contexts (such as inside a web server. For any long−running code, use one of the more verbose forms.

Of course, an iterator that steps through all of the records in an index is not usually what you want. The **iterator()** method accepts arguments that specify matching and sorting criteria, making it possible to construct iterators that return a subset of an index in a specified order.

The **where_clause** => <**sql−where**> argument matches a conditional phrase against the index's fields to narrow down the records that are returned. The **order_by** => <**sql−order−by**> argument controls the ordering of the records.

The **iterator()** method constructs a complex SQL statement that, when executed by the database, selects the records that the iterator will include. If a **where_clause** or **order_by** argument is supplied when an iterator is constructed, that piece of SQL logic is integrated into the iterator's complete SQL statement. Given a knowledge of generic SQL, it is easy to write **where_clause** and **order_by** arguments −− simply treat each field as you would a column in the database and the iterator parser will do the rest. An iterator with botha **where_clause** and an **order_by** might look like this:

```
# find all users with hotmail addresses and sort alphabetically
my $i = $index−>iterator ( where_clause => 'email LIKE "%.hotmail.com"',
                              order_by => 'email' );
```

Let's add another couple of fields to this index, so we can build some more interesting iterators. The *username* element is easy to add; it's just another field. If we also want to add the *zip* of the first *address*, that's a little harder. The index fields that we've seen so far map to top−level pieces of a Doc. One way to get at the *zip* information we need is to add a method to the `User` Def that fetches the *zip* of the first *address*. Here is the new method, along with the expanded index:

```
<!-- a method to return the zip code of the first address -->
<method>
  <name>first_zip</name>
  <code>
    <![CDATA[ sub { return $_[0]->element('address')->element('zip') } ]]>
  </code>
</method>

<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
</index>
```

Another way, if this method seems not likely to be used except to build the index table, is to add a **code** specifier to the field:

```
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field>
  <name>first_zip</name>
  <code>
    <![CDATA[ sub { return $_[0]->element('address')->element('zip') } ]]>
  </code>
</field>
</index>
```

As you can see, this is almost like adding another method to the Def −− in fact, we didn't change the embedded perl at all. The main difference is that we're not "cluttering up" the top level of our Def with a method that will only be used as part of the indexing operations. The **code** block is passed two arguments, the **doc** being indexed and the **index** element. It turns out that you almost always use the doc, and almost never use the index.

Adding a **code** block to a **field** disassociates the name of the field from the data that it stores. This is, obviously, useful. It can also be confusing. The default, non−code behavior is worth sticking to whenever possible, to keep defs and programs as maintainable as possible. (A **code** block can also be part of **collection** and **sort** elements, which are described below.)

And here are a few possible iterators:

```
# find all the users with .edu addresses -- in any order
my $i = $index->iterator ( where_clause => 'email LIKE "%.edu"' );

# find all the users with .edu addresses in Bevery Hills
my $i = $index->iterator ( where_clause => 'email LIKE "%.edu" AND zip = "90210'" );
```

```
# sort the .edu email addresses by string length in descending order, then
# alphabetically by username (uses mysql's LENGTH function)
my $i = $index->iterator ( where_clause => 'email LIKE "%.edu"',
                           order_by => 'DESC LENGTH(email), username' );
```

Plural Items in Indexes

The **field** elements of an index hold values derived from a Doc's elements and methods. As we've seen, fields can be used to select sets of records from an index, and to control the order in which those results are returned. One limitation of using fields in this way, however, is that each field can only hold a single value per record. Looked at another way, fields do an excellent job standing in for singular elements, but are not at all suited to dealing with plural elements. A field that corresponds to a plural element will always contain only the value of the first of those elements.

Another type of index element, the **collection**, is designed to accomodate plural values, and to allow the kinds of "sorting" operations that are common to many kinds of documents. Unlike a field, a collection cannot be used in a where clause; collections are a special−purpose tool. Let's add a collection to our `User` index that will allow us to select all of our records that include an address with a given zip−code −− any address this time, not just the first one.

```
<!−− a method to return the zip codes of each address, as an array −−>
<method>
  <name>zips</name>
  <code>
    <![CDATA[ sub {
      my @addresses;
      foreach my $addr $_[0]−>elements('address') {
        push @addresses, $addr−>element('zip')−>get();
      }
      return @addresses;
    ]]>
  </code>
</method>

<!−− the expanded index definition, now including the zips info
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
<collection><name>zips</name></collection></index>
```

We tied the zips collection to a method, but we could just as easily have tied it to a plural, non−nested element. The collection isn't particular, it just wants to be handed an array when the index is updated.

To select all of the users with an address in a given zip code, we request an iterator qualified by a **collection_spec**. A **collection_spec** is a string of the form **<collection_name>:<value>**. The **collection_spec** argument can be combined with the **where_clause** and **order_by** specifiers that we've already seen:

```
# select all the users with an address in the 20003 zip code -- in any order
my $i = $index−>iterator ( collection_spec => 'zips:20003' );

# select the users as above, and order them alphabetically by username
my $i = $index−>iterator ( collection_spec => 'zips:20003',
                           order_by => 'username' );
```

```
# select all of the users with an address in 20003 who also have a .edu
#   email address, and order them alphabetically by username
my $i = $index->iterator ( collection_spec => 'zips:20003',
                           where_clause => 'email LIKE "%.edu"',
                           order_by => 'username' );
```

Complex Collection Selectors

It is possible to specify complex collection_spec arguments, when creating an Iterator.

- The "pairs" in a spec can be strung together with `AND` and `OR`.
- AND'ed and OR'ed phrases can be parenthesized.
- Any pair can be prefixed with a `NOT` to ask for Docs that do not match the pair.
- A pair that includes spaces in its value part can be specified by surrounding the pair with single quotes.
- A pair that includes single quotes in its value part can be specified by surrounding the pair with single quotes and escaping the internal single quotes with a single backslash.

For example:

```
# select all of the users with an address in any 2000x OR 0213x zip code
# who also have a .edu email address, and order them alphabetically by
# username
my $i = $index->iterator ( collection_spec => 'zips:2000% AND zips:0213%',
                           where_clause => 'email LIKE "%.edu"',
                           order_by => 'username' );

# select users with an address NOT in the 20003 zip code.
my $i = $index->iterator ( collection_spec => 'NOT zips:20003' );

# a hypothetical collection pair with spaces and single quotes. We use
two backslashes here because the double quotes that surround the whole
string treat a single backslash as part of an escape character!
my $i = $index->iterator ( collection_spec => "'test:it\\'s easy' OR
                                               'test:it\\'s hard'" );
```

It is fairly easy to create complex specs that slow down database queries quite a lot. In particular, OR'ing together selections on large collections is very slow.

Full Text Search

A special content−holder is available that enables full−text search on an index component. We could make all of a `User`'s address information searchable by defining a method to generate a chunk of "address text", then defining an index **textsearch** container:

```
<!-- addresses_text: a method to glob all of a User's addresses together into
     a single string -->
<method>
<name>addresses_text</name>
<code>
  <![CDATA[
    sub {
      my $self = shift();
      my $addr_text = $self->full_name() . "\n";
      foreach my $a ( $self->elements('address') );
```

```
        $addr_text .= $a−>street1() . "\n".
        $addr_text .= $a−>street2() . "\n" .
        $addr_text .= $a−>city() . ' ' . $a−>state() . ' ' . $a−>zip() . "\n";
      }
      return $addr_text;
    }
  ]]>
</code>
</method>

<!−− the 'main' index, redefined to add full−text search on the addresses −−>
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
<collection><name>zips</name></collection>
<textsearch><name>addresses_text</name></textsearch>
</index>
```

With the new 'addresses_text' **textsearch** in place, we can use the full−text search feature in constructing iterators. A **textsearch_spec** argument specifies keywords that must appear in a record for it to be returned as part of an iterator's result set:

```
# look up all users who have the word "elm" (or "elms", "elmy", "elmed",
# etc.) in any of their addresses
my $i = $index−>iterator ( textsearch_spec=>'addresses_text:elm' );

# look up all users with an 'elm' and a 'springfield' in any
# of their addresses
my $i = $index−>iterator ( textsearch_spec=>'addresses_text:elm springfield' );
```

As the comments in the above example imply, the textsearch subsystem includes a "preprocessor" interface that allows words to be stemmed and pruned before indexing. The preprocessor defaults to **XML::Comma::Pkg::Textsearch::Preprocessor_En**, which handles English text. It includes a stop list of roughly 500 words, and relies on the CPAN module **Lingua::Stem** to do its stemming.

There are currently two other preprocessors in the standard distribution, **Preprocessor_Fr** for French and **Preprocessor_Sp** for Spanish. If you are only handling English−language content, you can skip the next three code examples, which detail how to specify Preprocessors other than **Preprocessor_En**.

A textsearch's **which_preprocessor** element controls which preprocessor will be used: **which_preprocessor** should define a sub that will be passed some combination of four arguments −− an active $doc; $index and $textsearch objects; and a search "attribute." The sub must return the name of the Preprocessor package that should be used. Here is a typical example:

```
  <textsearch>
    <name>body_text</name>
    <which_preprocessor>
      sub { return 'XML::Comma::Pkg::Textsearch::Preprocessor_Fr'; }
    </which_preprocessor>
  </textsearch>
```

Not much to it.

Things get somewhat more complex when we have to choose between multiple pre−processors on the fly. A Preprocessor is used in two different contexts: 1) when a doc is indexed, and 2) when a search is performed.

The routine below uses the $doc argument to determine what Preprocessor to use in the former case, and the $attribute argument in the latter.

```
  <textsearch>
    <name>paragraph</name>
    <which_preprocessor>
      <![CDATA[
        use XML::Comma::Pkg::Textsearch::Preprocessor_En;
        use XML::Comma::Pkg::Textsearch::Preprocessor_Sp;
        sub {
          my ( $doc, $index, $ts, $attribute ) = @_;
          if ( $doc−>lang_code() eq 'sp'  or  $attribute eq 'sp' ) {
            return 'XML::Comma::Pkg::Textsearch::Preprocessor_Sp';
          } else {
            return 'XML::Comma::Pkg::Textsearch::Preprocessor_En';
          }
        }
      ]]>
    </which_preprocessor>
  </textsearch>
```

The **$attribute** argument's value comes from the **textsearch_spec**, which has a special form for just this purpose:

```
# look up a word in the index, stemmed by the Spanish pre_processor
my $i = $index−>iterator ( textsearch_spec=>'body_text{sp}:lobos' );
```

The extra bit of text after the textsearch name, enclosed in curly brackets, is stripped off and passed as the **$attribute** to the **which_preprocessor** sub.

The back end of the **textsearch** facility is currently implemented on top of, and as part of, the Comma database−specific modules. It's efficiency is only mediocre, and performing the indexing operation on each document write is somewhat resource−intensive. Because of this, a **textsearch** can specify that it's operations should be deferred −− performed as a batch rather than on each and every update of the index. A cron job or application hook can be written to call an index's **sync_deferred_textsearches()** method at some convenient time (or at some regular interval).

```
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
<collection><name>zips</name></collection>
<textsearch>
  <name>addresses_text</name>
  <defer_on_update>1</defer_on_update>
</textsearch>
</index>
```

A number of important features are missing from the current implementation of full−text search: support for more languages, the ability to search for phrases within text, boolean OR'ing, etc. The strengths of the current implementation are that the full−text search is fully integrated with the rest of the database system, so complex iterators that include several different kinds of qualifiers can easily be constructed; and that the storage overhead is relatively small (only the inverted index is stored in the database, and that in a compressed form).

Work to improve the **textsearch** framework is certainly an area of interest for the Comma developers. It is likely that integration with database−provided full−text search capabilities is the best long−term option for fast, robust operation. Oracle certainly provides such capabilities. For the moment, the open source databases lag behind in this area.

Using an Iterator Over and Over: iterator_refresh()

It is often convenient to "reuse" an iterator. The **iterator_refresh()** method re−fills and resets the iterator. In its no−argument form **iterator_refresh()** is equivalent to asking the index for a new iterator with exactly the same specifications. However, the method also takes two optional arguments to limit the total number and the starting position of the results that are returned. Here are some examples:

```
# usage: $iterator−>iterator_refresh ( [ limit_number [, limit_offset ]] );

## simple refresh of a once−used iterator
#
my $i = $index−>iterator ( collection_spec => 'zips:20003' );
while ( $i++ ) {
  # ... do some stuff
}
$i−>iterator_refresh();
# now we can loop through again
while ( $i++ ) {
  # .. do some other stuff
}


## using iterator_refresh() to process only the first 10 results of a set
#
my $i = $index−>iterator()−>iterator_refresh ( 10 );
while ( $i++ ) {
  # ... do something with the first 10 (or fewer, if there weren't
  #     even that many)
}


## using iterator_refresh() to process the eleventh through fifteenth
## results (noting that the second argument, the offset, is zero−indexed)
#
my $i = $index−>iterator()−>iterator_refresh ( 5, 10 );
while ( $i++ ) {
  # ... do something with these five results (again, assuming that
  #     there are that many)
}
```

Fetching the Record's Doc: read_doc() and retrieve_doc()

Generally, an index should be designed so that its fields hold the most commonly−used pieces of information in a Doc. Of course, any criterion that will be used to select from an index must be available as a field or collection. Additionally, any part of a Doc that is regularly used during an iteration should also be defined as a field.

But sometimes you actually need to get the Doc itself from an iterator −− perhaps to do some complex read operation, or to check the content of an element that is so infrequently used that it makes little sense to include it as a field, or even to change the Doc and re−store it. Two iterator methods make this possible: **read_doc()**

and **retrieve_doc()**.

As the name suggests, **read_doc()** is analagous to **Doc−>read()** and fetches a read−only copy of the document, while **retrieve_doc()** is like **Doc−>retrieve()**, returning a fully modifiable Doc.

```
# print a simple list indicating how many addresses each User has defined
my $i = $index−>iterator();
while ( $i++ ) {
  print $i−>doc_key() . ': ' . scalar @{$i−>read_doc()−>elements('address')} . "\n";
}

# permanently delete (from the store that this index is tied to)
#   all documents with a .edu email address
my $i = $index−>iterator ( where_clause => 'email LIKE
"%.edu"' ); while ( $i++ ) {
  print "deleting " $i−>doc_key();
  $i−>retrieve_doc()−>erase();
}
```

Fetching One Record: single() and Company

In iterator retrieves a set of records, in a particular order. Sometimes you only want one record from an index. The **single()** method accepts the same arguments as **iterator()**, but it never returns more than one record, and if no records satisfy its specification it returns `undef`.

Usually, **single()** is used when you know there will only be one record in the index that matches your selection criteria. For example, we could write a **pre_store_hook** to make sure that no document is ever stored that has the same email address as a document that is already present. (See the section on advanced store techniques for more information about store hooks.)

```
<pre_store_hook>
<![CDATA[
  sub {
    my ( $self, $store ) = @_;
    my $email = $self−>element('email')−>get();
    my $index = $self−>def()−>get_index('main');
    if ( $index−>single(where_clause => "email = '$email'") ) {
      die "the email address '$email' is already in use\n";
    }
  }
]]>
</pre_store_hook>
```

The **single()** method isn't strictly necessary (you can always substitute some equivalent, if longer and more involved, iterator creation and refresh statement), but it does save some typing and makes code more readable. In the same spirit, two more methods exist that provide additional short−cuttage: **single_read()** and **single_retrieve()**.

As their names suggest, **single_read()** is a **single()** call plus (if possible) a **read_doc()**, and **single_retrieve** is the same except with **retrieve_doc()**. Both methods return `undef` if there is no record in the index that matches the iterator criteria. We can (somewhat frivolously) modify our **pre_store_hook** to demonstrate the use of **single_read()**:

```
<pre_store_hook>
<![CDATA[
  sub {
    my ( $self, $store ) = @_;
    my $email = $self->element('email')->get();
    my $index = $self->def()->get_index('main');
    if ( my $other_user = $index->single_read(where_clause => "email = '$email'") ) {
      my $other_users_location = $other_user->element('address')->[0]->element('zip')->get();
      die "the email address '$email' is already in use by someone in zip code $other_users_locat
    }
  }
]]>
</pre_store_hook>
```

Getting a Total Rather Than an Iterator: count()

One more index method exists that, like **single()**, **single_read()** and **single_retrieve()**, accepts the same arguments as **iterator()**: **count()**. The **count()** method returns the total number of records that match the supplied criteria.

```
# how many Users have a .edu email address?
my $total = $index->count ( where_clause => 'email LIKE "%.edu"' );
```

Using SQL Aggregates: aggregate (function => ...)

It is possible to get information from an Index using the "aggregate" functions provided by SQL. What functions are available (and how they work) are highly database−dependent. But, in general, it is possible to ask a Comma Index to create an **aggregate()** object in much the same way as an **iterator()** is created.

```
# ask for the sum of all the 'mailings_sent' columns in a
# hypothetical index
my $sum = $index->aggregate ( function => "SUM(mailings_sent)" );

# ask for the average age of all users in the 20003 zip code
my $avg_age = $index->aggregate ( function => "AVG(age)",
                                  collection_spec->'zips:20003' );
```

Actions at Index Time: index_hook

An **index_hook** is a hook that is run each time a record is added or updated, just before any changes are made to the database. Any number of these hooks can be defined for an index; they will be run in the order in which they appear in the Def. Two arguments are passed to the **index_hook** when it is invoked: the Doc being indexed and the index object. The return value of the hook is not used, but if the hook `dies` then the indexing operation is silently aborted.

Here is our index definition with a hook that prevents any ".edu" users from appearing in an index:

```
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
<collection><name>zips</name></collection>

<index_hook>
  <![CDATA[ sub {
      my ( $doc, $index ) = @_;
      die  if  $doc−>element('email')−>get() =~ /\.edu$/;
  } ]]>
</index_hook>
</index>
```

Defining Methods for an Index

You can add some flexibility to an index by defining **index methods**, pieces of code that can be called from an Iterator in the same way that element methods can be called from a Doc or element. Index methods are written for much the same reasons as element methods: to "standardize" a commonly−used piece of functionality, or to process information in a way that depends on some dynamic input (such as the time of day).

Here is our index definition with a fairly trivial index method added that simply checks whether a record's email address is in a hard−coded list:

```
<index>
<name>main</name>
<field><name>email</name></field>
<field><name>username</name></field>
<field><name>first_zip</name></field>
<collection><name>zips</name></collection>

<method>
  <name>in_denied_list</name>
  <code>
    <![CDATA[ sub {
          my @denied_list = ( 'spammer@spamming−org.org',
                              'impolite_person@unresponsize_isp.com' );
      my ( $iterator, @rest_of_args ) = @_;
      foreach my $denied ( @denied_list ) {
        return 1  if  $iterator−>email() eq $denied;
      }
      return 0;
    } ]]>
  </code>
</method>
</index>
```

And here is a simple scan through the entire index, checking if each record is in the denied list or not:

```
my $i = $index−>iterator();
while ( $i++ ) {
  if ( $i−>in_denied_list() ) {
    print $i−>username() . " is in the denied−list\n";
  } else {
    print $i−>username() . " is clean as a whistle\n";
  }
}
```

More Configuration, More SQL

An index is stored, behind the scenes, as a set of tables in a relational database. The Comma indexing framework provides a layer of abstraction on top of the database, and most of the time a programmer doesn't need to worry about the underlying implementation. But there are a few limitations that one should be aware of, and a few configuration directives that can make indexes more efficient and usable.

Configuring Fields

Each piece of information stored in a database must have a *type*. Every field in a Comma index defaults to the **VARCHAR(255)** type. It is necessary to specify a different type if a field needs to

- accomodate values longer than 255 characters, or
- use a more space−efficient representation, or
- support comparisons or operations (in where clauses, for example) other than those that VARCHAR provides.

A **sql_type** element can be used to specify the SQL type for a field. Here is an index definition with every field fully specified as to type:

```
<index>
<name>main</name>
<field>
  <name>email</name>
  <sql_type>VARCHAR(150)</sql_type>
</field>
<field>
  <name>username</name>
  <sql_type>VARCHAR(40)</sql_type>
</field>
<field>
  <name>first_zip</name>
  <sql_type>CHAR(5)</sql_type>
</field>
<collection><name>zips</name></collection>
</index>
```

The **doc_id_sql_type** element can be used to specify the SQL type for the columns that store the **doc_id** parts of records. This also defaults to **VARCHAR(255)**, and that's usually fine, but for particular efficiency it too can be changed. **NOTE:** unlike most other index elements, the **doc_id_sql_type** for an index should not be altered after an index is created the first time. If you need to change the **doc_id_sql_type**, you'll need to drop and rebuild the index. Collections can also specify a **sql_type**, which will be used only by 'binary table' type collections. Like the **doc_id_sql_type**, this specifier should not be changed for an existing index.

Another rarely−used specifier is the **store** element. By convention, an index shares its name with the store to which it is bound (remember, an index can only hold information about documents from a single store). But the convention is sometimes too restrictive −− for example, two indexes using the same store obviously can't have the same name. The **store** element allows an index to explicitly state with which store it is associated. Here is an index definition that uses both the **store** and **doc_id_sql_type** specifiers:

```
<index>
<name>just_emails</name>
<store>main</store>
<doc_id_sql_type>CHAR(12)</doc_id_sql_type>
<field>
  <name>email</name>
  <sql_type>VARCHAR(150)</sql_type>
</field>
</index>
```

The doc_id specifiers are passed directly to the relational database. Databases differ in how they define (and name) even commonly−used types, so which types are available from Comma will obviously depend on which database is being used. The Comma database adapters handle typing issues for the columns that are used internally as part of the abstraction layer, but when you start specifying SQL types for document ids and index fields, you're on your own.

Configuring Collections

Collection information can be stored in one of three ways:

- in a **stringified** form in the main Index table (the default)
- in a **binary table** containing a list of doc_id/value pairs
- in **many tables**, each containing a list of doc_ids

Each of these has advantages and disadvantages.

The **stringified** type is the simplest. No extra database tables are created, and the values of the collection are available directly from the Iterator. Partial matches (using the SQL wildcard character **%**) that are anchored at either the front or back of a string are allowed.

The **binary table** type requires one additional table to be maintained in the database. This type of collection has the potential to provide the best mixture of flexibility and speed, although the current implementation is less advanced than it should be. A number of optimizations that would improve the speed of this type of storage are yet to be performed, and the MySQL database doesn't support some types of queries that would be particularly useful in this regard. Partial matches are allowed, but NOT pairs are not. And any given binary−table−typed collection may be used only once collection_spec.

The **many tables** type requires a table to maintained in the database for each unique value that occurs in the collection. Collections of this type are usually tied to elements that are defined by an **enum** (or are similarly restricted in their allowed values).

This type is not very flexible, but can provide the quickest select times for many applications. Because **many tables** data is spread across several database tables, it is often possible for the system to take shortcuts that are not possible for **stringified** or **binary table** collection. And each of the tables in a collection of this type can be **clean()**ed separately (see below for an explanation of `<clean>` clauses), further reducing the amount of data that must be sifted through to create each Iterator. Partial matches are not allowed for many−tables−typed collections, but NOT pairs are.

Here are three collections that will hold the same data but store that data differently behind the scenes (with <clean> behavior specified for the **many tables** version, for good measure:

```
<collection>
  <name>zips_str</name>
  <code><![CDATA[ sub { $_[0]->zips() } ]]></code>
</collection>

<collection>
  <name>zips_bt</name>
  <type>binary table</type>
  <code><![CDATA[ sub { $_[0]->zips() } ]]></code>
</collection>


<collection>
  <name>zips_mt</name>
```

```
      <type>many tables</type>
      <code><![CDATA[ sub { $_[0]->zips() } ]]></code>
      <clean>
        <to_size>10_000</to_size>
        <size_trigger>10_500</size_trigger>
        <order_by>doc_id DESC</order_by>
      </clean>
    </collection>
```

The fields=> Argument

One more way to increase the speed of Iterator creation is to ask for only a subset of the fields defined in the Index:

```
# we only need email addresses
my $iterator = $index->iterator ( fields => [ 'email' ] );
```

This reduces the amount of data that the database must shuffle around when joining tables together. It generally only helps when used in conjunction with many−tables− and binary−table−type collections, and on Indexes that define many fields. But in some cases the speed increases can be dramatic: 20% or more.

The `fields=>` argument should take the form of a reference to an array of field names. The resulting Iterator will only have data for the `doc_id` and `record_last_modified` pseudo−fields, and for the fields named in the argument array ref.

Changes: Automatic Updating of Database Structure

Comma does its best to adjust the table structures of the database to match any changes that are made to an index definition. Mostly, this is possible. You can always add or remove a collection or field. You can usually change the SQL type of a field −− only the willingness of the database to convert already−stored information from the old type to the new limits this.

You **can not** change the **doc_id_sql_type** without dropping and rebuilding the database.

Clean and Rebuild

Left to their own devices, indexes will grow as new records are added. But larger indexes can be slower to query, or take up too much disk space. And some indexes are designed specifically to contain only a subset of documents: the *n* most recently−updated, for example, or all of the documents that haven't yet been tagged "archive."

You have already seen how to write an **index_hook** that prevents documents from being added to the index. But to dynamically scan an index and remove records requires a different approach.

The **clean()** method triggers the "cleaning" of an index. During a clean, records are deleted according to the criteria listed in the index definition's **clean** section. Here is a simple example index definition that uses a clean section to delete records that haven't been "used" in 30 days:

```
<index>
  <name>main</name>
  <field>username</field>
  <field>email</field>
  <field>last_used</field>
  <clean>
    <erase_where_clause>last_used > 60*60*24*30</erase_where_clause>
  </clean>
</index>
```

The **clean** method can, of course, be called just like any other, but it is common to use a cron job to clean a database each night, or each week.

```
# simply call clean() to clean an index
$index−>clean()

# perhaps a command−line version from within a cron (or similar) job
perl −MXML::Comma −e 'XML::Comma::Def−>read(name=>"User")−>get_index("main")−>clean();';
```

The clean operation is as careful as possible about running in the background: records added while a clean is in progress are ignored by the clean, and only one clean runs at a time (if **clean()** is called while another clean is already in progress, it simply returns).

The **erase_where_clause** is one style of clean, but there is another. A clean can define a **to_size** element to trim an index down to a certain number of records, and specify an **order_by** clause to make sure that the list of records is arranged in the correct order before being trimmed. (If no **order_by** is given, the index's **default_order_by** is used, which itself defaults to **doc_id**.) Here is a clean section that keeps only the 1000 most−recently−used records:

```
<clean>
  <to_size>1000</to_size>
  <order_by>last_used DESC</order_by>
</clean>
```

An alternative to calling **clean()** manually (or from a cron job) is to configure cleaning to take place automatically when an index contains a certain number of records. If a **size_trigger** element is present, Comma will check the size of the index after each **insert()** and if that size equals or exceeds the number given in the **size_trigger** specifier then a clean will be triggered.

```
<clean>
  <to_size>1000</to_size>
  <order_by>last_used DESC</order_by>
  <size_trigger>1200</size_trigger>
</clean>
```

A **clean** section can also apply to an individual collection, rather than to the entire index. In this case, the clean effects only a table of pointers holding collection information, not the main records themselves. Configuring an index so that its collection tables stay reasonably small can dramatically improve performance.

Sometimes, even a clean isn't clean enough. If several fields are added to an existing index, or the criteria for indexing changes radically, or a problem with an index is identified, it can be necessary to **rebuild()** the index.

A rebuild can be done starting from scratch (after dropping any existing index tables), or on a fully−populated index. In either case, the rebuild will add and update information in the database while it runs, then clean the index after it is finished −− so an index can be used while a rebuild is in progress. Care should be taken, however, not to stop a rebuild operation before it has completed.

As might be expected, the **rebuild()** method is used to trigger a rebuild operation. During a rebuild, Comma iterates backwards through all of the documents in a store, calling **index_update()** on each Doc. All of the normal rules apply, so Docs that would not be added to an index by an explicit call to **index_update()** are not added by a **rebuild()**, and all hooks are run as normal for each Doc.

It can take a long time to handle all of the documents in a large store, and often an index will only want to treat a subset of documents. The **stop_rebuild_hook** allows some "stop−now−if" logic to be inserted into the rebuild cycle. This hook was designed to gracefully handle the common case of an index that does not include documents that are "older" than a certain cut−off age. (Which is also, of course, why the rebuild cycle iterates *backwards* through a store. To be of much use, a **stop_rebuild_hook** must be used in conjunction with a store that sorts doc ids by criteria that are roughly similar to the criteria that the **stop_rebuild_hook** cares about. But you already knew that.)

The **stop_rebuild_hook** is passed two arguments, the doc in question and the index object, and should return true if the rebuild should stop cycling through the storage documents and move on to its cleanup phase.

```
<index>
  <name>new_users</name>
  <store>sequential_user_id_store</store>
  <field>username</field>
  <field>email</field>
  <field>created_timestamp</field>
  <stop_rebuild_hook>
    <![CDATA[ sub {
        my ( $doc, $index ) = @_;
        my $age = time() − $doc−>created_timestamp();
        return $age > (60*60*24*90);
      } ]]>
  </stop_rebuild_hook>
</index>
```

Storage in More Detail: Hooks, Output Filters and Location Modules

Hooks: pre_store_hook, post_store_hook, erase_hook

Three types of hooks are available to run during store operations. Just before a document is stored, each defined **pre_store_hook** runs. Three arguments are passed: the doc being stored and (though these are rarely needed) the store definition and a hashref containing the args that were passed to the original store() call. (This access to the original %args allows hook code to take actions based on programmer−defined args. You can pass anything you want as part of the store args −− its completely open−ended.) If any of the **pre_store_hooks** `die` then the store operation is aborted, the remaining hooks are ignored, and a `STORE_ERROR` is thrown.

Just after a doc is stored, each defined **post_store_hook** runs, again taking the doc in question, the store definition and the args hashref as its arguments. Every **post_store_hook** runs and any errors that are thrown are ignored.

As a side note: very occasionally you may want to modify the doc itself inside a **post_store_hook**, and then to save those changes. (You might do this if, for example, you need to have the doc's id available before your code can run, which requires that the code be installed as a post rather than pre store_hook.) A special flag for the **store()** method −− **no_hooks=>1** −− is available to allow a **store()** to be performed without its attendant pre and post hooks. This is obviously a power that should not be abused.

Before a doc is erased, each defined **erase_hook** runs. Three arguments are passed to an **erase_hook**: the doc being erased, the store definition, and the doc's doc_location. If any **erase_hook** dies the erase operation is aborted (and no more hooks are run). It's worth noting that an erase operation happens −− and the erase_hooks are run −− during a **move()** as well as during a simple **erase()**.

More on Location Chains

A store definition's location chain controls how a doc is written out to long−term storage. A location chain must generate both a storage "location" and a document id. The current location modules all use the filesytem, but −− in principle at least −− the Store interface is abstract enough that modules could equally well use a database, a tape drive, or a networked archive of some kind.

Location modules come in two kinds, which betray their file−system−centric roots by being called *_dir* and *_file*. The _dir modules specify "directory" locations; a dir module cannot be used as the final link in a location chain. The _file modules specify the "file" portion of a location; a _file module can only be used as the final link in a chain.

Standard _dir Modules

Sequential_dir creates sequentially−numbered directories. It takes one argument, which is optional: **max** specifies the highest legal number in the sequence, and defaults to 9999. Once the **Sequential_dir** becomes "full" −− the **max** number has been reached −− all subsequent store attempts throw an error. The directory name will be padded with leading zeros to make alpha−numeric sorting possible (note that if the **max** specifier is changed the width of subsequently−created directory names could change, possibly ruining the sort). The id fragment generated by the **Sequential_dir** module is simply the directory name.

GMT_3layer_dir creates a directory structure derived from the current date. As the name implies, three directory layers are created, in the pattern YYYY/MM/DD. The id fragment that this module generates is the directory structure without internal separators: YYYYMMDD.

Derived_dir creates a directory from the contents of a doc element or the return value of a method. The **derive_from** argument is required, and specifies the element or method that will be called to generate the filename −− this works like the shortcut syntax: `$doc−><derive_from>()`. The **width** argument is also required, and specifies the number of characters that will be in the directory name. The **derive_from**'ed value will be left−padded with zeros if it is shorter than **width**, or truncated if it is longer. The id fragment generated by **Derived_dir** is the same as the directory name.

Standard _file Modules

Sequential_file creates unique, sequentially−numbered files. It takes two arguments, which are both optional. The **max** argument specifies the highest legal number in the sequence, and defaults to 9999. Once the **Sequential_file** becomes "full" all subsequent store attempts throw an error. The filenames are padded with leading zeros up to the width of the **max** number. (As is apparent, **Sequential_file** operates very much along the same lines as **Sequential_dir**.) An additional argument, **extension**, specifies what extension should be added to the filename. The extension should include the separator character (a period is the most common separator), and defaults to `.comma`. The id fragment that **Sequential_file** generates is the filename, stripped of its extension.

Derived_file creates a filename from the contents of a doc element or the return value of a method. The **derive_from** argument is required, and specifies the element or method that will be called to generate the filename −− this works like the shortcut syntax: `$doc−><derive_from>()`. The optional **extension** argument defaults to `.comma`. Here is an example:

```
<store>
  <name>main</name>
  <location>Derived_file: 'derive_from','foo', 'extension', '.xml'</location>
</store>
```

Read_only_file is used for collections of documents that will be by some non−Comma tool. Parts of Comma use **Read_only_file** to read config files, which are always edited by hand. Trying to write to a location chain that uses **Read_only_file** will result in an error. The optional **extension** argument specifies the extention that is present on the files, and defaults to `.comma`. Here is the definition used by *HTTP_Upload_Config*, part of the *HTTP_Upload* package:

```
<store>
  <name>main</name>
  <base>config</base>
  <location>Read_only_file:'extension','.config'</location>
</store>
```

Output Filters

Like location chains, output chains influence how a document is written to long−term storage. Whereas a location chain determines "where" a document is stored, an output chain determines the "format" of the stored doc.

There is an implicit output format present for every single document: plain text. Document storage always starts with the generation of a plain−text, xml−marked−up representation of the doc (the same thing that is produced by a call to **to_string()**), and document retrieval always ends with the parsing of that plain text representation. But various output filters can be added to the storage/retrieval process, with each one influencing what bytes actually get written out to disk.

The **Gzip** output filter compresses the doc using the gzip algorithm. Large documents can be compressed quite effectively. **Gzip** takes no arguments and operates as simply as possible. Other tools that understand gzip compression/decompression −− `zcat`, for example −− can be used to examine or process the stored docs independent of Comma.

```
<store>
  <name>gzipped</name>
  <location>Sequential_file</location>
  <output>Gzip</output>
</store>
```

The **Twofish** filter uses the Twofish symmetric encryption algorithm (implemented by Abhijit Menon−Sen's Crypt::Twofish module) to encrypt the storage output. **Twofish** needs an argument, **key**, specifying the encryption/decryption key to be used. An additional argument, **key_hash**, is also required. The **key_hash** is used to verify that the key supplied is correct. This is important because encrypting data with a mis−typed (or otherwise wrongly−supplied) key can cause much heartache and difficulty. (And keys can −− and likely should −− be dynamically supplied, so there is often opportunity to mis−type a key.)

The **key_hash** is produced by generating an md5 digest of the key, and should be supplied as a hex string. The following one−liner spits out the **key_hash** for the **key** 'foo':

```
perl −MDigest::MD5 −e 'print Digest::MD5::md5_hex("foo") . "\n"'
```

And here is an example of a store that first gzips, then Twofish encrypts, its docs:

```
<store>
  <name>gzipped_and_encrypted</name>
  <location>Sequential_file</location>
  <output>Gzip</output>
  <output>Twofish: 'key', 'an encryption p',
                         'key_hash', '67d8db90c079ae74967a1b750b87525f'</output>
</store>
```

The **HMAC_MD5** output filter uses Gisle Aas's Digest::HMAC_MD5 module to fingerprint each of its stored docs. Like the **Twofish** filter, **HMAC_MD5** requires **key** and **key_hash** arguments. (The **key_hash** is generated in the same way as for **Twofish**.) Here is a store that gzips, Twofish encrypts and HMACs its docs:

```
<store>
  <name>five</name>
  <base>gz_hmac_twofish</base>
  <location>Sequential_file:'max',10,'extension','.gz_hmac_encrypt'</location>

  <output>Gzip</output>

  <output>HMAC_MD5: 'key',      'an−hmac−sillykey',
                    'key_hash', '7c116a20dcc378de2afb4cc9955a2187'</output>

  <output>Twofish: 'key',      'another−sillykey',
                   'key_hash', '6ae8eaeaa226a03a46d79a359ab00db0'</output>
</store>
```

As mentioned above, keys will often need to be supplied when Comma applications are loaded, rather than hard−coded into defs. (Leaving the key in the def in plain text is a potential security problem.) Here is a toy storage definition that prompts the user to enter the key on the command line when the def is loaded:

```
<store>
  <name>four</name>
  <base>test/four</base>
  <location>Sequential_file:'max',10,'extension','.encrypt'</location>
  <output>
   <![CDATA[
   Twofish:
     'key' => do { print "key ('1234'): "; my $key=<>; chop $key; $key },
     'key_hash' => '81dc9bdb52d04dc20036dbd8313ed055'
  ]]>
  </output>
</store>
```

Writing New Output and Location Modules

Hmmm. See the text file: Storage/Location/location_modules.doc for basic location module theory and practice. Output filters are much simpler −− use the source, Luke.

Grouping and Sorting Elements

Comma provides a pair of methods that rearrange the order of elements in a doc or nested element.

Prettifying: group_elements()

Calling the **group_elements()** method of a doc or nested element pulls together all sub−elements of the same type, and arranges these groupings of elements according to the order in which they are listed in the doc or nested element's def. The order of the elements in each group relative to one another remains unchanged. Often this method is used to "prettify" a doc so that it will be more easily readable or hand−editable. The **group_elements()** method returns the object on which it was called.

```
<!-- a simple def -->
<DocumentDefinition>
  <name>group_test</name>
  <element><name>a</name></element>
  <element><name>b</name></element>
  <plural>'a','b'</plural>
</DocumentDefinition>

<!-- and a doc -->
<group_test>
<a>0</a>
<b>1</b>
<a>2</a>
<b>3</b>
</group_test>

# assume the above is '$doc' -- call group_elements and to_string() from some
# code...
print $doc->group_elements()->to_string();

# will print out:
<group_test>
<a>0</a>
<a>2</a>
<b>1</b>
<b>3</b>
</group_test>
```

Sorting: sort_elements()

The **sort_elements()** method is much more flexible and powerful. When called with no arguments, it rearranges all of the elements in a container according to a defined **sort_sub**. Like **elements()**, it accepts an optional list of tags as arguments; if called with arguments it rearranges only the specified types of elements.

The **sort_sub** is specified in one of two places: **sort_elements()** first looks in the definition of the first callee for a **sort_sub**, then in the definition of the caller. If it doesn't find a **sort_sub** in either def, the method throws an error. The **sort_elements()** method (again, for cognitive compatibility with **elements()** returns a sorted array −− or reference −− of elements).

Here is a document definition showing four slightly different uses of a **sort_sub**. The *simple* element's sort would be controlled by the document's **sort_sub** (which appears near the end of the definition), as it doesn't have one of its own. The other three elements all define **sort_sub** elements that (presumably) are tailored for the types of data they will contain and the ways in which they will be used.

```
<DocumentDefinition>
  <name>sort_test</name>

  <element><name>simple</name></element>

  <element>
    <name>self_sorting_alpha</name>
    <sort_sub><![CDATA[ sub ($$) { $_[1]->get() cmp $_[0]->get(); } ]]>
  </element>

  <element>
    <name>self_sorting_numeric</name>
    <sort_sub><![CDATA[ sub ($$) { $_[1]->get() <=> $_[0]->get(); } ]]>
  </element>

  <nested_element>
    <name>self_sorting_nested</name>
    <element><name>rank</name></element>
    <sort_sub><![CDATA[ sub ($$) { $_[0]->rank() <=> $_[1]->rank(); } ]]>
  </nested_element>

  <plural>'simple','self_sorting_alpha','self_sorting_numeric','self_sorting_nested'</plural>
  <sort_sub><![CDATA[ sub ($$) { $_[0]->get() <=> $_[1]->get(); } ]]>
</DocumentDefinition>
```

Each **sort_sub** string is turned into a code reference (by an eval statement), then passed to a `sort` statement whenever **sort_elements()** is called. Note that you must use the *prototyped* form of subroutine definition for the sort statement to work properly. You may be more used to using the special variables `$a` and `$b` −− the `($$)` prototype simply tells `sort` to use normal subroutine parameters instead: `$_[0]` and `$_[1]`.

```
# the above def in use
my $doc = XML::Comma::Doc−>new ( type=>'sort_test' );
$doc−>add_element('simple')−>(1);
$doc−>add_element('simple')−>(23);
$doc−>add_element('simple')−>(11);
$doc−>add_element('self_sorting_alpha')−>('ccc');
$doc−>add_element('self_sorting_alpha')−>('aaa');
$doc−>add_element('self_sorting_alpha')−>('bbb');
$doc−>add_element('self_sorting_numeric')−>(10);
$doc−>add_element('self_sorting_numeric')−>(11);
$doc−>add_element('self_sorting_numeric')−>(3);
$doc−>add_element('self_sorting_nested')−>ranked(7);
$doc−>add_element('self_sorting_nested')−>ranked(5);
$doc−>add_element('self_sorting_nested')−>ranked(10);

# do some sorts
$doc−>sort_elements ( 'simple' );
$doc−>sort_elements ( 'self_sorting_alpha' );
$doc−>sort_elements ( 'self_sorting_numeric' );

# do a sort and actually use the return value
print join "\n", map { "rank of " . $_ . ": " . $_−>rank() }
  $doc−>sort_elements ( 'self_sorting_nested' );
```

Error Handling and Logging

Comma includes error propogation facilities and some basic logging functionality. The **log_file** configuration variable specifies a file for Comma to log to.

Each line in the log file is made up of three fields, separated by spaces: 1) the unix time, 2) the pid, and 3) the error string. Most errors that are thrown as part of Comma's internal workings will have an error string made up a standard error name, then two dashes, then more information about the error, then the file and line number of the caller. For example:

```
1000838058 1584 STORE_ERROR −− no store given to first−time Doc−>store() at t/storage.t line 167
```

Two public methods enable writing to the log file. **XML::Comma::Log−>err()** takes an error name and a description string as arguments, writes a line to the log, then exits with a `die`. The error name should be a short, arbritrary string which contains no spaces and identifies the error. All errors thrown by internal comma code use all−caps error names, which makes the log files easy to read, but that's only a convention. The description string can be as long as desired, and should contain more specific information about the error (newline characters will be replaced with spaces when the string is written to the log file.) A file and line−number from which the **err()** method was called will be included in the output.

XML::Comma::Log−>warn() takes a string, appends the text `WARNING −−` to it, and writes a line to the log. It doesn't `die` or record the file and line−number of the caller.

```
# non−fatal, informational logging
XML::Comma::Log->warn ( "whoa, we might have problems" );

# throw a fatal error
XML::Comma::Log−>err ( "FAKE_ERROR", "we have broken down" );
```

Network Transfer

Comma ships with a client/server library for transferring Docs across the network. The tranfer operations are built on top of the HTTP protocol, and the server side of the library is designed to run as a mod_perl handler inside Apache. The client side can be used programmatically, just like any other part of the Comma system.

Here is a bit of example code:

```
my $t = XML::Comma::Pkg::Transfer::HTTP_Transfer->new
  ( target          => 'https://remote.server.com/util/transfer' );

if ( ! $t->ping() ) {
  die "couldn't contact the remote server";
}

my $key = 'article|main|0012';
my $article = XML::Comma::Doc->read ( $key );
if ( $article->comma_hash ne $t->get_hash($key) ) {
  print "transferring: $key ... ";
  $t->put ( $article );
  print "ok\n";
} else {
  print "hash matched for $key";
}
```

Client Methods

The client is an instance of the **XML::Comma::Pkg::Transfer::HTTP_Transfer** class. The constructor for this class takes one optional argument, `target`, which specifies the URL to which the client will connect. The available client methods are: **ping()**, **put()**, **put_push()**, **get_hash()** and **get_and_store()**.

The **ping()** tests the connection to the server. It returns `1` if the client is able to exchange data with the server and returns `undef` otherwise.

The **put()** and **put_push()** methods transfer docs from the client to the server. Both methods take a doc object as their argument. A doc that is **put()** across the network is stored on the server using the same **store** and **id** as the local doc. (This implies that a newly−created and not−yet−stored doc may not be **put()**ted.) A doc that is **put_push()**ed across the network will be stored on the server in a new location −− as if it were newly−created. The **put_push()** method takes an optional second argument, a **store_name** indicating what store to use on the server. If that argument is omitted, the doc's store is used. Both methods return the doc id under which the doc was stored (though this is presumably already known in the **put()** case). Both methods throw an error if they encounter unrecoverable network difficulties or problems on the remote server.

The **get_hash()** method gets a document's **comma_hash** from the remote server. The method takes the three "retrieval" arguments −− either paramaterized as a **type**=>, **store**=> and **id**=> or stringified as a **key**. A call to **get_hash()** returns the key on success, returns `undef` if the requested doc is not found on the remote server, and throws an error if it encounters severe difficulties across the network.

The **get_and_store()** method takes the same "retrieval" arguments as **get()**, but pulls the remote doc across and stores it on the local server. It returns a read−only copy of the doc on success, and throws an error on failure.

The *HTTP::Transfer* library is not designed to support transferring documents bi−directionally within a single store. All kinds of potential problems *will* arise if one attempts to do that.

Server <Location /util/transfer> Configuration

To bring up an *HTTP_Transfer* server, an Apache handler must be configured for a particular **<Location>**. The following code shold be all that is required for a basic installation.

```
<Location /util/transfer>
  SetHandler perl−script
  PerlHandler XML::Comma::Pkg::Transfer::HTTP_Transfer
</Location>
```

Access Control and SSL Encryption

Apache's extensive access control facilities can be used to control usage of an *HTTP_Transfer* server. The Apache documentation describes how to limit access by connection−oriented criteria such as IP address.

The *HTTP_Transfer* library is also SSL−aware, allowing data to be sent across the network in an encrypted form. Apache must be configured with SSL support for this option to be available. On the client side, all that is required is to specify a target url that begins with *https*.

Client SSL certificates can be used to further limit access to the server. Apache must be configured so that it has access to a "Certificate Authority" public certificate and with the following two SSL options:

```
SSLVerifyClient require
SSLVerifyDepth  1
```

This will limit access to clients that hold certificates signed by the "Certificate Authority's" private key. The following two constructor arguments can be used to make a certificate/key pair available to the *HTTP_Transfer* client.

```
# use client certificate:
    https_cert_file => '/tmp/cert.pem'
    https_key_file  => '/tmp/key.pem'

# example
my $t = XML::Comma::Pkg::Transfer::HTTP_Transfer−>new
  ( target          => 'https://remote.server.com/util/transfer',
    https_cert_file => '/path/cert.pem',
    https_key_file  => '/path/key.pem' );
```

The key file should not be passphrase encrypted. The Crypt::SSLeay library that *HTTP_Transfer* relies on does not cache the key, so the passphrase must be re−typed on each connection if an encrypted key is used.

Reference: Defs

Document definitions, which are Comma documents like any other, are themselves constrained by a definition. This "bootstrap" definition is often useful as a reference.

```
<DocumentDefinition>
  <name>DocumentDefinition</name>

  <element><name>name</name></element>
  <element><name>ignore_for_hash</name></element>
  <element><name>plural</name></element>
  <element><name>required</name></element>

  <element><name>validate_hook</name></element>
  <element><name>document_write_hook</name></element>
  <element><name>def_hook</name></element>
  <element><name>sort_sub</name></element>

  <nested_element>
    <name>method</name>
    <element><name>name</name></element>
    <element><name>code</name></element>
    <required>'name','code'</required>
  </nested_element>

  <nested_element>
    <name>element</name>
    <element><name>name</name></element>
    <element><name>validate_hook</name></element>
    <element><name>set_hook</name></element>
    <element><name>default</name></element>
    <element><name>macro</name></element>
    <element><name>defname</name></element>
    <element><name>sort_sub</name></element>
    <plural>'validate_hook','set_hook','macro'</plural>
    <required>'name'</required>
  </nested_element>

  <nested_element>
    <name>blob_element</name>
    <element><name>name</name></element>
    <element><name>extension</name></element>
  </nested_element>

  <nested_element>
    <name>nested_element</name>
    <element><name>name</name></element>
    <element><name>defname</name></element>

    <element><name>macro</name></element>
    <element><name>plural</name></element>
    <element><name>required</name></element>
    <element><name>ignore_for_hash</name></element>
    <element><name>validate_hook</name></element>
    <element><name>sort_sub</name></element>

    <nested_element>
      <name>element</name>
      <defname>DocumentDefinition:element</defname>
    </nested_element>
    <nested_element>
      <name>blob_element</name>
      <defname>DocumentDefinition:blob_element</defname>
    </nested_element>

    <nested_element>
      <name>nested_element</name>
      <defname>DocumentDefinition:nested_element</defname>
    </nested_element>

    <nested_element>
      <name>method</name>
      <defname>DocumentDefinition:method</defname>
    </nested_element>

    <plural>
      'macro',
      'plural',
      'required',
      'ignore_for_hash',
      'validate_hook',
      'element',
      'blob_element',
      'nested_element',
      'method',
    </plural>
    <required>'name'</required>
  </nested_element>

  <nested_element>
    <name>store</name>
    <element><name>name</name></element>
    <element><name>location</name></element>
    <element><name>output</name></element>
    <element><name>root</name></element>
    <element><name>base</name></element>
    <element>
      <name>file_permissions</name>
      <default>664</default>
    </element>
    <element><name>pre_store_hook</name></element>
    <element><name>post_store_hook</name></element>
    <element><name>erase_hook</name></element>
    <element><name>index_on_store</name></element>
    <plural>qw( location           output
                pre_store_hook     post_store_hook
                erase_hook
                index_on_store  )</plural>
    <required>'name','base','location'</required>
  </nested_element>

  <nested_element>
    <name>index</name>
    <element><name>name</name></element>
    <!-- 'store' will default to self->element('name')->get() -
         (note, this is handled by the Store->store() method in
          the internals ) -->
    <element><name>store</name></element>
    <!-- doc_id_sql_type SHOULD NOT BE CHANGED without completely
         dropping and recreating a given index's database (or otherwise
         altering the database structure outside of Comma). ** there is
```

```
     no automatic change of this to match a def ** -->
<element>
  <name>doc_id_sql_type</name>
  <default>VARCHAR(255)</default>
</element>
<nested_element>
  <name>field</name>
  <element><name>name</name></element>
  <element><name>code</name></element>
  <element>
    <name>sql_type</name>
    <default>VARCHAR(255)</default>
  </element>
  <required>'name'</required>
</nested_element>
<nested_element>
  <name>collection</name>
  <element><name>name</name></element>
  <element><name>code</name></element>
</nested_element>
<nested_element>
  <name>sort</name>
  <element><name>name</name></element>
  <element><name>code</name></element>
  <nested_element>
    <name>clean</name>
    <defname>DocumentDefinition:index:clean</defname>
  </nested_element>
  <required>'name'</required>
</nested_element>
<nested_element>
  <name>textsearch</name>
  <element><name>name</name></element>
  <element><name>defer_on_update</name></element>
  <required>'name'</required>
</nested_element>
<nested_element>
  <name>sql_index</name>
  <element><name>name</name></element>
  <element><name>unique</name></element>
  <element><name>fields</name></element>
  <required>'name','fields'</required>
</nested_element>
<element>
  <name>default_order_by</name>
  <default>doc_id</default>
</element>
<nested_element>
  <name>order_by_expression</name>
  <element><name>name</name></element>
  <element><name>expression</name></element>
  <required>'name','expression'</required>
</nested_element>
<nested_element>
  <name>clean</name>
  <element><name>to_size</name></element>
  <element><name>order_by</name></element>
  <element><name>size_trigger</name></element>
  <element><name>erase_where_clause</name></element>
</nested_element>
<element><name>index_hook</name></element>
<element><name>stop_rebuild_hook</name></element>
```

```
    <nested_element>
      <name>method</name>
      <defname>DocumentDefinition:method</defname>
    </nested_element>
    <plural>qw( field
                collection
                sort
                textsearch
                sql_index
                order_by_expression
                index_hook
                stop_rebuild_hook
                method )</plural>
    <required>'name'</required>
  </nested_element>

  <plural>
            'element',
            'nested_element',
            'blob_element',
            'method',
            'store',
            'index',
            'document_write_hook',
            'plural',
            'required',
            'ignore_for_hash',
            'validate_hook',
  </plural>

</DocumentDefinition>
```

Reference: Hooks

def_hook −− any def. The **def_hook** is unlike the other hooks in its operation. Rather than being defined as part of a document definition and run during later operations, the **def_hook** is run as part of the loading of the def itself. This hook is designed to allow a def to create dynamic structures that will be available to its various methods, hooks and elements. The contents of the def_hook element should be a block of code suitable for `eval`ing. Any error thrown during the `eval` will cause a `DEF_HOOK_ERR` to be propogated.

document_write_hook ($doc) −− any doc. The **to_string()** method triggers the execution of any defined hooks of this type, and any error thrown by one of these hooks will abort the to_string operation. This hook is passed the doc that it is attached to as its only argument.

erase_hook ($doc, $store, $location) −− any store. This hook is run at the beginning of an erase operation (triggered by doc **erase()** and **move()** methods). Any thrown errors abort the erase operation and cause a `STORE_ERROR` to be propogated. This hook is passed the doc being erased, the store object, and the doc location.

index_hook ($doc, $index) −− any index. This hook is run at the beginning of an **index_update()**. If an **index_hook** throws an error, the update halts and the call to **index_update()** returns `undef` (no error is propogated, however). This hook is passed the doc that is the source of the update and the index object as its two arguments.

post_store_hook ($doc, $store, \%store_arguments) −− any store. This hook is run at the very end of a store operation (triggered by doc **store()**, **move()**, and **copy()** methods). All hooks are run, then the doc is unlocked, then −− if necessary −− the first of any errors that might have been encountered as the hooks were run is thrown. This hook is passed the doc being stored and the store object as its arguments.

pre_store_hook ($doc, $store, \%store_arguments) −− any store. This type of hook is run at the very beginning of a store operation (triggered by doc **store()**, **move()**, and **copy()** methods). If any **pre_store_hook** throws an error, the store operation is aborted and a `STORE_ERROR` is propogated. This hook is passed the doc being stored and the store object as its arguments.

stop_rebuild_hook ($doc, $index) −− any index. This hook is run during an index **rebuild()**, after each doc is processed. The hook should return a true value to indicate that the rebuild operation has completed (and, conversely, a false value to indicate that the rebuild should continue). This hook is passed the doc that −− if the process continues −− will next be added to the index and the index object itself as its two arguments.

validate_hook ($element, [$content]) −− any nested element, non−nested element, or doc. The element **validate()**, **validate_content()**, **set()**, and **append()** methods trigger the execution of these hooks, as does the nested element **validate()** method. If any **validate_hook()** throws an error, the validate operation is halted and a `BAD_CONTENT` or `VALIDATE_ERROR` is propogated. The hook is passed one or two arguments: the element to which the hook is attached and (for non−nested elements) the content to be validated.

set_hook ($element, $content_reference, \%set_arguments) −− non−nested elements and blob elements. This hook is triggered by a call to an element's **set()** method. Any **set_hook**s defined for an element is run after any **validate_hook**s and before the actual set operation takes place. A **set_hook** receives, as its second argument, a reference to the content that was passed to the **set()** method, enabling the hook to modify the content, if necessary. If any **set_hook** dies, an error is thrown and the set is aborted.

set_from_file_hook ($element, $filename, \%set_arguments) −− blob elements. This hook is exactly analagous to the **set_hook** described above, except that it is unique to blob elements, is triggered by the blob element **set_from_file()** method, and is passed the filename that **set_from_file()** receives instead of a content reference. If you want to intercept all "set" operations on a blob element, you must define both of these hooks.

read_hook() −− any doc or element. This hook is called when a doc or element is "read in" from storage, during a **read()** or **retrieve()**. After the system creates and initializes the element (including any sub−elements or content that are read in), any defined **read_hook**s are called. No arguments are passed. This hook is sometimes needed as a complement to a **set_hook**, which is only called by a **set()** method invocation and not upon reading an element in from storage. Note that a **read_hook** can only be called on an element that "exists" in the stored doc: empty elements aren't stored, and so they can't be read in and hook'ed.

Reference: Perl API (Methods and Objects)

- XML::Comma
  - <all configuration variables readable via methods>
  - $lock_interface = XML::Comma−>lock_singlet()
    - undef = $lock_interface−>wait_for_hold ($string);
    - undef = $lock_interface−>release_hold ($string);
- XML::Comma::Doc
  - $doc = XML::Comma::Doc−>new (type =>)
  - $doc = XML::Comma::Doc−>new (block =>)
  - $doc = XML::Comma::Doc−>new (file =>)
  - $doc = XML::Comma::Doc−>retrieve (key, [timeout=><seconds>])
  - $doc = XML::Comma::Doc−>retrieve (store =>, type =>, id =>, [timeout=><seconds>])
  - $doc || undef = XML::Comma::Doc−>retrieve_no_wait (key)
  - $doc || undef = XML::Comma::Doc−>retrieve_no_wait (store =>, type =>, id =>)
  - $doc = XML::Comma::Doc−>read (key)
  - $doc = XML::Comma::Doc−>read (<retrieve arguments>)
  - $doc = $doc−>get_lock ([timeout=><seconds>]);
  - $doc || undef = $doc−>get_lock_no_wait();
  - $string = $doc−>to_string()
  - $string = $doc−>comma_hash()
  - $self = $doc−>store (store=>, [keep_open=>], [no_hooks=>], [args...])
  - $self = $doc−>erase()
  - $self = $doc−>copy()
  - $self = $doc−>copy() (<store arguments>)
  - $self = $doc−>move()
  - $self = $doc−>move() (<store arguments>)
  - $store = $doc−>doc_store()
  - $string = $doc−>doc_location()
  - $string = $doc−>doc_id()
  - $string = $doc−>doc_key()
  - $string = $doc−>doc_source_file()
  - $bool = $doc−>doc_is_locked()
  - $int = $doc−>doc_last_modified()
  - $doc = $doc−>index_update (index=>$index)
  - $doc = $doc−>index_remove (index=>$index)
- all elements
  - $string = $el−>tag()
  - $string = $el−>tag_up_path()
  - $def = $el−>def()
  - $return_val = $el−>method ($name, [@args...])
  - null = $el−>set_attr ($name => $value, [$name => $value ...]);
  - $string = $el−>get_attr ($name);
  - $hash_ref = $def−>def_pnotes();
  - @names = $def−>applied_macros();
  - 1/undef = $def−>applied_macros (@names);
- blob elements
  - $string = $el−>set($string)
  - $string = $el−>get()

    - " = $el−>set_from_file ($filename)
    - " = $el−>validate()
    - $string = $el−>get_location()
- simple elements
    - $string = $el−>get([unescape=>], [%args])
    - $string = $el−>get_without_default()
    - $string = $el−>set ($string, [escape=>], [%args])
    - $string = $el−>append ($more_string)
    - $string = $el−>validate()
    - $string = $el−>validate_content ($string)
    - 1 = $el−>cdata_wrap();
- nested elements
    - @els/[] = $el−>elements ([@tags])
    - $el = $el−>element ($tag)
    - $el = $el−>add_element ($tag)
    - $el = $el−>delete_element ($tag)
    - @strings/[] = $el−>elements_group_get ($tag)
    - @strings/[] = $el−>elements_group_add ($tag, @strings)
    - @els/[] = $el−>elements_group_delete ($tag, @strings)
    - $bool = $el−>elements_group_lists ($tag, $string)
    - $bool = $el−>element_is_plural ($tag)
    - $bool = $el−>element_is_defined ($tag)
    - $bool = $el−>element_is_nested ($tag)
    - $bool = $el−>element_is_blob ($tag)
    - $bool = $el−>element_is_required ($tag)
    - " = $el−>validate()
    - [DEPRECATED] " = $el−>validate_structure()
    - @els = $el−>get_all_blobs()
    - $el = $el−>group_elements();
    - @els/[] = $el−>sort_elements ([@tags])
- XML::Comma::Def
    - $def = XML::Comma::Def−>read (name =>)
    - @names = $def−>store_names();
    - $store = $def−>get_store ($name);
    - @names = $def−>index_names();
    - @names = $def−>method_names();
    - $store = $def−>get_index ($name);
    - $hash_ref = $def−>def_pnotes();
    - $code_ref = $def−>add_hook ($hook_type, $string || $code_ref);
    - $code_ref = $def−>add_method ($name, $string || $code_ref);
    - $code_ref || undef = $def−>method_code ($name);
    - @return/[] = $def−>method ($name, @args);
    - @names = $def−>applied_macros();
    - 1/undef = $def−>applied_macros (@names);
    - @els/[] = $def−>def_sub_elements();
    - $el = $def−>def_by_name ($element_name);
    - 1/undef = $def−>is_required();
    - 1/undef = $def−>is_plural();
    - 1/undef = $def−>is_nested();
    - 1/undef = $def−>is_blob();
    - 1/undef = $def−>is_ignore_for_hash();

- XML::Comma::Indexing::Index
    - @names = $index−>field_names();
    - @names = $index−>sort_names(); [DEPRECATED]
    - @names = $index−>collection_names();
    - @names = $index−>textsearch_names();
    - @names = $index−>method_names();
    - $type_name = $index−>collection_type ($collection_name);
    - $iterator = $index−>iterator ([%args]);
    - $iterator/undef = $index−>single ([%args]);
    - $doc/undef = $index−>single_read ([%args]);
    - $doc/undef = $index−>single_retrieve ([%args]);
    - $int = $index−>count ([%args]);
    - $int = $index−>last_modified_time ([$sort_name, $sort_string]);
    - $val = $index−>aggregate (function=> [%args]);
    - " = $index−>rebuild ([verbose=>'1'|'0',workers=>$processes_num,size=>$size_num]);
    - " = $index−>clean();
    - " = $index−>get_dbh();
- XML::Comma::Indexing::Iterator
    - $iterator = $iterator_refresh() ([$limit_number], [$limit_offset]);
    - $iterator/false = $iterator−>iterator_next();
    - $bool = $iterator−>iterator_has_stuff();
    - $string = $iterator−>iterator_select_returnval();
    - $doc = $iterator−>retrieve_doc();
    - $doc = $iterator−>read_doc();
    - $string = $iterator−>doc_key();
    - $string = $iterator−>doc_id();
    - $return_value = $iterator−>$field/$method ([@args]);
- XML::Comma::Log
    - <thrown error/die> = XML::Comma::Log−>err ($error_string, $info_string);
    - " = XML::Comma::Log−>warn ($string);
    - " = XML::Comma::Log−>log ($string/$error);
- XML::Comma::Storage::Store
    - $id_string = $store−>first_id();
    - $id_string = $store−>last_id();
    - $id_string = $store−>next_id ($id_string);
    - $id_string = $store−>prev_id ($id_string);
    - $directory = $store−>base_directory();
- XML::Comma::Storage::Iterator
    - $iterator = XML::Comma::Storage::Iterator−>new (store=>$store, size=>$num, pos=><'+' | '−'>;
    - $num = $iterator−>length();
    - $num = $iterator−>index();
    - $num = $iterator−>set($num);
    - $num = $iterator−>next_id();
    - $num = $iterator−>prev_id();
    - $num = $iterator−>next_retrieve();
    - $num = $iterator−>prev_retrieve();
    - $num = $iterator−>next_read();
    - $num = $iterator−>prev_read();
    - $num = $iterator−>doc_id();
- XML::Comma::Util

- $first_element = trim (@strings_to_trim);
- @trimmed_strings = trim (@strings_to_trim);
- $bool = array_includes (@array, $string);
- @array/[] = arrayref_remove_dups ($array_ref);
- @array/[] = arrayref_remove ($array_ref, @els/[]);
- @array = flatten_arrayrefs (@arrays/[] [...]);
- $escaped_string = XML_basic_escape ($string);
- $unescaped_string = XML_basic_unescape ($string);
- $escaped_string = XML_smart_escape ($string); # ignores entities
- $escaped_string = XML_bare_amp_unescape ($string); # ditto
- '' = dbg (@arrays/[] [...]);
- ($name, @args) = name_and_args_eval ($string);
- $string = random_an_string ($length);

Appendix: Table Structure of Index Databases

Each Comma **index** creates (and uses) at least one table in the SQL database. All of these tables are kept track of by an *index_tables* table. It is usually possible for a programmer or system administrator to remain blissfully ignorant of the information presented here. This section is written for the curious and the unlucky.

The index_tables Table

The *index_tables* table contains a record for each database table that has been created as part of an index's backing store. The fields are as follows:

- _comma_flag
- _sq
- doctype
- index_name
- table_name
- table_type
- last_modified
- sort_spec
- textsearch
- collection
- index_def

The *_comma_flag* field is used by parts of the system that need to mark a table as in use. The **rebuild()** method, for example, tags any tables that it is working on, and will refuse to begin work if any tables for an index appear to be so tagged.

The *_sq* field is a unique, ascending integer sequence, and is used to generate unique names for all the tables that Comma creates.

The *doctype* and *index_name* fields together identify which Comma index a table "belongs" to.

The *table_name* field gives the name of the table. When a new table is created, a name is generated by appending an underscore and the next valid *_sq* integer to the first few letters of the doctype.

The *table_type* field is an integer indicating what kind of table this is (see below).

The *last_modified* field indicates when a table was last changed, but is currently not much used.

Only one of the *sort_spec*, *textsearch*, *collection* and *index_def* fields is used by any single record: these fields hold extra information relevant to the various kinds of tables.

Table Type 1: The Data Table

Every index has a Data table. Each row in the data table represents a single record in the index. The table has three standard columns: `_comma_flag` holds a status value, `_sq` holds a short, unique identifier that can be used to refer back to this record, and `doc_id` the doc_id of the document this record is drawn from. The rest of the columns in the table are created from the fields and collections defined by the index; each is named the same as the field, typed according to the field's **sql_type**, and holds the contents of that field or collection for the record in question. Field columns simply hold the scalar value returned by the field's element or method call. Collection columns hold a string consisting of all of the values returned by the element or method call, each blocked between two 'pipe' characters, concatenated all together.

Table Type 2: The 'many tables' Table

An index may have as many 'many tables' collection tables as the database permits. Each table is created on demand when an update encounters a new value in the collection.

Each table contains only two columns, the familiar `_comma_flag` field used for coordintation and locking by various pieces of the indexing code, and a `doc_id` column. The table simply keeps track of the documents that belong in that sort; logical joins are used to select subsets of records that match sort criteria.

Table Types 3 and 4: Textsearch Index and Defers Tables

Each textsearch defined by an index uses two tables. The main table, called a textsearch_index_table by the system internals, stores an inverted index, with each record in the table mapping a word to a packed array containing data table _sq keys. The second table −− the textsearch_defers table −− contains a list of actions that have been performed on the index since the last **sync_deferred_textsearches()** call.

Table Type 5: The 'binary table' Table

A 'binary table' collection table is created for each binary−table−typed collection in each Index. Each of these tables contains three columns, the familiarl `_comma_flag` field, a `doc_id` field and a `value` field. The table maps doc_id/value pairs together, so that a where'd selection can be followed by a logical join to determine subsets of records that match sort criteria.