ZConfig Package Reference

Release 2.1

Zope Corporation

April 12, 2004

Lafayette Technology Center
513 Prince Edward Street
Fredericksburg, VA 22401
http://www.zope.com/

Abstract

This document describes the syntax and API used in configuration files for components of a Zope installation written by Zope Corporation. This configuration mechanism is itself configured using a schema specification written in XML.

Contents

1 Introduction

Zope uses a common syntax and API for configuration files designed for software components written by Zope Corporation. Third-party software which is also part of a Zope installation may use a different syntax, though any software is welcome to use the syntax used by Zope Corporation. Any software written in Python is free to use the `ZConfig` software to load such configuration files in order to ensure compatibility. This software is covered by the Zope Public License, version 2.0.

The `ZConfig` package has been tested with Python 2.3. Older versions of Python are not supported. `ZConfig` only relies on the Python standard library.

Configurations which use `ZConfig` are described using *schema*. A schema is a specification for the allowed structure and content of the configuration. `ZConfig` schema are written using a small XML-based language. The schema language allows the schema author to specify the names of the keys allowed at the top level and within sections, to define the types of sections which may be used (and where), the types of each values, whether a key or section must be specified or is optional, default values for keys, and whether a value can be given only once or repeatedly.

2 Configuration Syntax

Like the `ConfigParser` format, this format supports key-value pairs arranged in sections. Unlike the `ConfigParser` format, sections are typed and can be organized hierarchically. Additional files may be included if needed. Schema components not specified in the application schema can be imported from the configuration file. Though both formats are substantially line-oriented, this format is more flexible.

The intent of supporting nested section is to allow setting up the configurations for loosely-associated components in a container. For example, each process running on a host might get its configuration section from that host's section of a shared configuration file.

The top level of a configuration file consists of a series of inclusions, key-value pairs, and sections.

Comments can be added on lines by themselves. A comment has a '#' as the first non-space character and extends to the end of the line:

```
# This is a comment
```

An inclusion is expressed like this:

```
%include defaults.conf
```

The resource to be included can be specified by a relative or absolute URL, resolved relative to the URL of the resource the `%include` directive is located in.

A key-value pair is expressed like this:

```
key value
```

The key may include any non-white characters except for parentheses. The value contains all the characters between the key and the end of the line, with surrounding whitespace removed.

Since comments must be on lines by themselves, the '#' character can be part of a value:

```
key value # still part of the value
```

Sections may be either empty or non-empty. An empty section may be used to provide an alias for another section.

A non-empty section starts with a header, contains configuration data on subsequent lines, and ends with a terminator.

The header for a non-empty section has this form (square brackets denote optional parts):

<*section-type* [*name*] >

section-type and *name* all have the same syntactic constraints as key names.

The terminator looks like this:

</*section-type*>

The configuration data in a non-empty section consists of a sequence of one or more key-value pairs and sections. For example:

```
<my-section>
    key-1 value-1
    key-2 value-2

    <another-section>
        key-3 value-3
    </another-section>
</my-section>
```

(The indentation is used here for clarity, but is not required for syntactic correctness.)

The header for empty sections is similar to that of non-empty sections, but there is no terminator:

<*section-type* [*name*] />

2.1 Extending the Configuration Schema

As we'll see in section 3, "Writing Configuration Schema," what can be written in a configuration is controlled by schemas which can be built from *components*. These components can also be used to extend the set of implementations of objects the application can handle. What this means when writing a configuration is that third-party implementations of application object types can be used wherever those application types are used in the configuration, if there's a `ZConfig` component available for that implementation.

The configuration file can use an `%import` directive to load a named component:

```
%import Products.Ape
```

The text to the right of the `%import` keyword must be the name of a Python package; the `ZConfig` component provided by that package will be loaded and incorporated into the schema being used to load the configuration file. After the import, section types defined in the component may be used in the configuration.

More detail is needed for this to really make sense.

A schema may define section types which are *abstract*; these cannot be used directly in a configuration, but multiple concrete section types can be defined which *implement* the abstract types. Wherever the application allows an abstract type to be used, any concrete type which implements that abstract type can be used in an actual configuration.

The `%import` directive allows loading schema components which provide alternate concrete section types which implement the abstract types defined by the application. This allows third-party implementations of abstract types to be used in place of or in addition to implementations provided with the application.

Consider an example application application which supports logging in the same way Zope 2 does. There are some parameters which configure the general behavior of the logging mechanism, and an arbitrary number of *log handlers* may be specified to control how the log messages are handled. Several log handlers are provided by the application. Here is an example logging configuration:

```
<eventlog>
  level verbose

  <logfile>
    path /var/log/myapp/events.log
  </logfile>
</eventlog>
```

A third-party extension may provide a log handler to send high-priority alerts the system administrator's text pager or SMS-capable phone. All that's needed is to install the implementation so it can be imported by Python, and modify the configuration:

```
%import my.pager.loghandler

<eventlog>
  level verbose

  <logfile>
    path /var/log/myapp/events.log
  </logfile>

  <pager>
    number   1-800-555-1234
    message  Something broke!
  </pager>
</eventlog>
```

2.2 Textual Substitution in Values

`ZConfig` provides a limited way to re-use portions of a value using simple string substitution. To use this facility, define named bits of replacement text using the `%define` directive, and reference these texts from values.

The syntax for `%define` is:

`%define` *name* [*value*]

The value of *name* must be a sequence of letters, digits, and underscores, and may not start with a digit; the namespace for these names is separate from the other namespaces used with `ZConfig`, and is case-insensitive. If *value* is omitted, it will be the empty string. If given, there must be whitespace between *name* and *value*; *value* will not include any whitespace on either side, just like values from key-value pairs.

Names must be defined before they are used, and may not be re-defined. All resources being parsed as part of a configuration share a single namespace for defined names. This means that resources which may be included more than once should not define any names.

References to defined names from configuration values use the syntax described for the `ZConfig.substitution` module. Configuration values which include a '`$`' as part of the actual value will need to use `$$` to get a single '`$`' in the result.

The values of defined names are processed in the same way as configuration values, and may contain references to named definitions.

For example, the value for `key` will evaluate to `value`:

```
%define name value
key $name
```

3 Writing Configuration Schema

`ZConfig` schema are written as XML documents.

Data types are searched in a special namespace defined by the data type registry. The default registry has slightly magical semantics: If the value can be matched to a standard data type when interpreted as a **basic-key**, the standard data type will be used. If that fails, the value must be a **dotted-name** containing at least one dot, and a conversion function will be sought using the `search()` method of the data type registry used to load the schema.

3.1 Schema Elements

For each element, the content model is shown, followed by a description of how the element is used, and then a list of the available attributes. For each attribute, the type of the value is given as either the name of a `ZConfig` datatype or an XML attribute value type. Familiarity with XML's Document Type Definition language is helpful.

The following elements are used to describe a schema:

<**schema**>
    `description?, metadefault?, example?, import*, (sectiontype | abstracttype)*, (section | key | multisection | multikey)*`
</**schema**>

Document element for a `ZConfig` schema.

`extends` (**space-separated-url-references**)
A list of URLs of base schemas from which this section type will inherit key, section, and section type declarations. If omitted, this schema is defined using only the keys, sections, and section types contained within the `schema` element.

`datatype` (**basic-key** or **dotted-name**)
The data type converter which will be applied to the value of this section. If the value is a **dotted-name** that

begins with a period, the value of `prefix` will be pre-pended, if set. If any base schemas are listed in the `extends` attribute, the default value for this attribute comes from the base schemas. If the base schemas all use the same `datatype`, then that data type will be the default value for the extending schema. If there are no base schemas, the default value is **null**, which means that the `ZConfig` section object will be used unconverted. If the base schemas have different `datatype` definitions, you must explicitly define the `datatype` in the extending schema.

`handler` (**basic-key**)

`keytype` (**basic-key** or **dotted-name**)

The data type converter which will be applied to keys found in this section. This can be used to constrain key values in different ways; two data types which may be especially useful are the **identifier** and **ipaddr-or-hostname** types. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. If any base schemas are listed in the `extends` attribute, the default value for this attribute comes from the base schemas. If the base schemas all use the same `keytype`, then that key type will be the default value for the extending schema. If there are no base schemas, the default value is **basic-key**. If the base schemas have different `keytype` definitions, you must explicitly define the `keytype` in the extending schema.

`prefix` (**dotted-name**)

Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts with the `schema` element if it hasn't been overridden by an inner element with a `prefix` attribute.

<**description**>
`PCDATA`
</**description**>

Descriptive text explaining the purpose the container of the `description` element. Most other elements can contain a `description` element as their first child. At most one `description` element may appear in a given context.

`format` (NMTOKEN)

Optional attribute that can be added to indicate what conventions are used to mark up the contained text. This is intended to serve as a hint for documentation extraction tools. Suggested values are:

Value	Content Format
`plain`	text/plain; blank lines separate paragraphs
`rest`	reStructuredText
`stx`	Classic Structured Text

<**example**>
`PCDATA`
</**example**>

An example value. This serves only as documentation.

<**metadefault**>
`PCDATA`
</**metadefault**>

A description of the default value, for human readers. This may include information about how a computed value is determined when the schema does not specify a default value.

<**abstracttype**>
`description?`
</**abstracttype**>

Define an abstract section type.

`name` (**basic-key**)

The name of the abstract section type; required.

<**sectiontype**>
`description?, (section | key | multisection | multikey)*`

</**sectiontype**>

Define a concrete section type.

datatype (**basic-key** or **dotted-name**)

The data type converter which will be applied to the value of this section. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. If `datatype` is omitted and `extends` is used, the `datatype` from the section type identified by the `extends` attribute is used.

extends (**basic-key**)

The name of a concrete section type from which this section type acquires all key and section declarations. This type does *not* automatically implement any abstract section type implemented by the named section type. If omitted, this section is defined with only the keys and sections contained within the `sectiontype` element. The new section type is called a *derived* section type, and the type named by this attribute is called the *base* type. Values for the `datatype` and `keytype` attributes are acquired from the base type if not specified.

implements (**basic-key**)

The name of an abstract section type which this concrete section type implements. If omitted, this section type does not implement any abstract type, and can only be used if it is specified directly in a schema or other section type.

keytype (**basic-key**)

The data type converter which will be applied to keys found in this section. This can be used to constrain key values in different ways; two data types which may be especially useful are the **identifier** and **ipaddr-or-hostname** types. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. The default value is **basic-key**. If `keytype` is omitted and `extends` is used, the `keytype` from the section type identified by the `extends` attribute is used.

name (**basic-key**)

The name of the section type; required.

prefix (**dotted-name**)

Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts in the `sectiontype` element. If omitted, the prefix specified by a containing context is used if specified.

<**import**>
`EMPTY`
</**import**>

Import a schema component. Exactly one of the attributes `package` and `src` must be specified.

file (file name without directory information)

Name of the component file within a package; if not specified, 'component.xml' is used. This may only be given when `package` is used. (The 'component.xml' file is always used when importing via `%import` from a configuration file.)

package (**dotted-suffix**)

Name of a Python package that contains the schema component being imported. The component will be loaded from the file identified by the `file` attribute, or 'component.xml' if `file` is not specified. If the package name given starts with a dot ('`.`'), the name used will be the current prefix and the value of this attribute concatenated.

src (**url-reference**)

URL to a separate schema which can provide useful types. The referenced resource must contain a schema, not a schema component. Section types defined or imported by the referenced schema are added to the schema containing the `import`; top-level keys and sections are ignored.

<**key**>
`description?, example?, metadefault?`
</**key**>

A `key` element is used to describe a key-value pair which may occur at most once in the section type or top-level schema in which it is listed.

attribute (**identifier**)
: The name of the Python attribute which this key should be the value of on a `SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the key name to underscores.

datatype (**basic-key** or **dotted-name**)
: The data type converter which will be applied to the value of this key. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set.

default (**string**)
: If the key-value pair is optional and this attribute is specified, the value of this attribute will be converted using the appropriate data type converter and returned to the application as the configured value. This attribute may not be specified if the `required` attribute is `yes`.

handler (**basic-key**)

name (**basic-key**)
: The name of the key, as it must be given in a configuration instance, or '`*`'. If the value is '`*`', any name not already specified as a key may be used, and the configuration value for the key will be a dictionary mapping from the key name to the value. In this case, the `attribute` attribute must be specified, and the data type for the key will be applied to each key which is found.

required (`yes|no`)
: Specifies whether the configuration instance is required to provide the key. If the value is `yes`, the `default` attribute may not be specified and an error will be reported if the configuration instance does not specify a value for the key. If the value is `no` (the default) and the configuration instance does not specify a value, the value reported to the application will be that specified by the `default` attribute, if given, or `None`.

<**multikey**>
`description?, example?, metadefault?, default*`
</**multikey**>

A `multikey` element is used to describe a key-value pair which may occur any number of times in the section type or top-level schema in which it is listed.

attribute (**identifier**)
: The name of the Python attribute which this key should be the value of on a `SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the key name to underscores.

datatype (**basic-key** or **dotted-name**)
: The data type converter which will be applied to the value of this key. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set.

handler (**basic-key**)

name (**basic-key**)
: The name of the key, as it must be given in a configuration instance, or '`+`'. If the value is '`+`', any name not already specified as a key may be used, and the configuration value for the key will be a dictionary mapping from the key name to the value. In this case, the `attribute` attribute must be specified, and the data type for the key will be applied to each key which is found.

required (`yes|no`)
: Specifies whether the configuration instance is required to provide the key. If the value is `yes`, no `default` elements may be specified and an error will be reported if the configuration instance does not specify at least one value for the key. If the value is `no` (the default) and the configuration instance does not specify a value, the value reported to the application will be a list containing one element for each `default` element specified as a child of the `multikey`. Each value will be individually converted according to the `datatype` attribute.

<**default**>
`PCDATA`

</**default**>

Each `default` element specifies a single default value for a `multikey`. This element can be repeated to produce a list of individual default values. The text contained in the element will be passed to the datatype conversion for the `multikey`.

key ()

Key to associate with the default value. This is only used for defaults of a `key` or `multikey` with a `name` of `+`; in that case this attribute is required.

<**section**>
    description?
</**section**>

A `section` element is used to describe a section which may occur at most once in the section type or top-level schema in which it is listed.

attribute (**identifier**)

The name of the Python attribute which this section should be the value of on a `SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the section name to underscores, in which case the `name` attribute may not be `*` or `+`.

handler (**basic-key**)

name (**basic-key**)

The name of the section, as it must be given in a configuration instance, `*`, or `+`. If the value is `*`, any name not already specified as a key may be used. If the value is `*` or `+`, the `attribute` attribute must be specified. If the value is `*`, any name is allowed, or the name may be omitted. If the value is `+`, any name is allowed, but some name must be provided.

required (`yes|no`)

Specifies whether the configuration instance is required to provide the section. If the value is `yes`, an error will be reported if the configuration instance does not include the section. If the value is `no` (the default) and the configuration instance does not include the section, the value reported to the application will be `None`.

type (**basic-key**)

The section type which matching sections must implement. If the value names an abstract section type, matching sections in the configuration file must be of a type which specifies that it implements the named abstract type. If the name identifies a concrete type, the section type must match exactly.

<**multisection**>
    description?
</**multisection**>

A `multisection` element is used to describe a section which may occur any number of times in the section type or top-level schema in which it is listed.

attribute (**identifier**)

The name of the Python attribute which matching sections should be the value of on a `SectionValue` instance. This is required and must be unique within the immediate contents of a section type or schema. The `SectionValue` instance will contain a list of matching sections.

handler (**basic-key**)

name (**basic-key**)

For a `multisection`, any name not already specified as a key may be used. If the value is `*` or `+`, the `attribute` attribute must be specified. If the value is `*`, any name is allowed, or the name may be omitted. If the value is `+`, any name is allowed, but some name must be provided. No other value for the `name` attribute is allowed for a `multisection`.

required (`yes|no`)

Specifies whether the configuration instance is required to provide at least one matching section. If the value is `yes`, an error will be reported if the configuration instance does not include the section. If the value is `no` (the default) and the configuration instance does not include the section, the value reported to the application will be `None`.

type (**basic-key**)
The section type which matching sections must implement. If the value names an abstract section type, matching sections in the configuration file must be of types which specify that they implement the named abstract type. If the name identifies a concrete type, the section type must match exactly.

3.2 Schema Components

XXX need more explanation

`ZConfig` supports schema components that can be provided by disparate components, and allows them to be knit together into concrete schema for applications. Components cannot add additional keys or sections in the application schema.

A schema *component* is allowed to define new abstract and section types. Components are identified using a dotted-name, similar to a Python module name. For example, one component may be `zodb.storage`.

Schema components are stored alongside application code since they directly reference datatype code. Schema components are provided by Python packages. The component definition is normally stored in the file 'component.xml'; an alternate filename may be specified using the `file` attribute of the `import` element. Components imported using the `%import` keyword from a configuration file must be named 'component.xml'. The component defines the types provided by that component; it must have a `component` element as the document element.

The following element is used as the document element for schema components. Note that schema components do not allow keys and sections to be added to the top-level of a schema; they serve only to provide type definitions.

<**component**>
`description?, (abstracttype | sectiontype)*`
</**component**>
The top-level element for schema components.

prefix (**dotted-name**)
Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts within the `component` element if it hasn't been overridden by an inner element with a `prefix` attribute.

4 Standard `ZConfig` Datatypes

There are a number of data types which can be identified using the `datatype` attribute on `key`, `sectiontype`, and `schema` elements. Applications may extend the set of datatypes by calling the `register()` method of the data type registry being used or by using Python dotted-names to refer to conversion routines defined in code.

The following data types are provided by the default type registry.

basic-key
The default data type for a key in a ZConfig configuration file. The result of conversion is always lower-case, and matches the regular expression ⌜`[a-z][-._a-z0-9]*`⌟.

boolean
Convert a human-friendly string to a boolean value. The names `yes`, `on`, and `true` convert to `True`, while `no`, `off`, and `false` convert to `False`. Comparisons are case-insensitive. All other input strings are disallowed.

byte-size
A specification of a size, with byte multiplier suffixes (for example, '`128MB`'). Suffixes are case insensitive and may be '`KB`', '`MB`', or '`GB`'

dotted-name
A string consisting of one or more **identifier** values separated by periods ('`.`').

dotted-suffix
A string consisting of one or more **identifier** values separated by periods ('`.`'), possibly prefixed by a period. This can be used to indicate a dotted name that may be specified relative to some base dotted name.

existing-dirpath
Validates that the directory portion of a pathname exists. For example, if the value provided is '/foo/bar', '/foo' must be an existing directory. No conversion is performed.

existing-directory
Validates that a directory by the given name exists on the local filesystem. No conversion is performed.

existing-file
Validates that a file by the given name exists. No conversion is performed.

existing-path
Validates that a path (file, directory, or symlink) by the given name exists on the local filesystem. No conversion is performed.

float
A Python float. `Inf`, `-Inf`, and `NaN` are not allowed.

identifier
Any valid Python identifier.

inet-address
An Internet address expressed as a `(`*hostname*`,` *port*`)` pair. If only the port is specified, an empty string will be returned for *hostname*. If the port is omitted, `None` will be returned for *port*.

integer
Convert a value to an integer. This will be a Python `int` if the value is in the range allowed by `int`, otherwise a Python `long` is returned.

ipaddr-or-hostname
Validates a valid IP address or hostname. If the first character is a digit, the value is assumed to be an IP address. If the first character is not a digit, the value is assumed to be a hostname. Hostnames are converted to lower case.

locale
Any valid locale specifier accepted by the available `locale.setlocale()` function. Be aware that only the `'C'` locale is supported on some platforms.

null
No conversion is performed; the value passed in is the value returned. This is the default data type for section values.

port-number
Returns a valid port number as an integer. Validity does not imply that any particular use may be made of the port, however. For example, port number lower than 1024 generally cannot be bound by non-root users.

socket-address
An address for a socket. The converted value is an object providing two attributes. `family` specifies the address family (`AF_INET` or `AF_UNIX`), with `None` instead of `AF_UNIX` on platforms that don't support it. The `address` attribute will be the address that should be passed to the socket's `bind()` method. If the family is `AF_UNIX`, the specific address will be a pathname; if the family is `AF_INET`, the second part will be the result of the **inet-address** conversion.

string

Returns the input value as a string. If the source is a Unicode string, this implies that it will be checked to be simple 7-bit ASCII. This is the default data type for values in configuration files.

time-interval

A specification of a time interval in seconds, with multiplier suffixes (for example, `12h`). Suffixes are case insensitive and may be '`s`' (seconds), '`m`' (minutes), '`h`' (hours), or '`d`' (days).

5 Standard `ZConfig` Schema Components

`ZConfig` provides a few convenient schema components as part of the package. These may be used directly or can server as examples for creating new components.

5.1 `ZConfig.components.basic`

The `ZConfig.components.basic` package provides small components that can be helpful in composing application-specific components and schema. There is no large functionality represented by this package. The default component provided by this package simply imports all of the smaller components. This can be imported using

```
<import package="ZConfig.components.basic"/>
```

Each of the smaller components is documented directly; importing these selectively can reduce the time it takes to load a schema slightly, and allows replacing the other basic components with alternate components (by using different imports that define the same type names) if desired.

The Mapping Section Type

There is a basic section type that behaves like a simple Python mapping; this can be imported directly using

```
<import package="ZConfig.components.basic" file="mapping.xml"/>
```

This defines a single section type, **ZConfig.basic.mapping**. When this is used, the section value is a Python dictionary mapping keys to string values.

This type is intended to be used by extending it in simple ways. The simplest is to create a new section type name that makes more sense for the application:

```
<import package="ZConfig.components.basic" file="mapping.xml"/>

<sectiontype name="my-mapping"
             extends="ZConfig.basic.mapping"
             />

<section name="*"
         type="my-mapping"
         attribute="map"
         />
```

This allows a configuration to contain a mapping from **basic-key** names to string values like this:

```
<my-mapping>
  This that
  and the other
</my-mapping>
```

The value of the configuration object's `map` attribute would then be the dictionary

```
{'this': 'that',
 'and': 'the other',
 }
```

(Recall that the **basic-key** data type converts everything to lower case.)

Perhaps a more interesting application of **ZConfig.basic.mapping** is using the derived type to override the `keytype`. If we have the conversion function:

```
def email_address(value):
    userid, hostname = value.split("@", 1)
    hostname = hostname.lower()  # normalize what we know we can
    return "%s@%s" % (userid, hostname)
```

then we can use this as the key type for a derived mapping type:

```
<import package="ZConfig.components.basic" file="mapping.xml"/>

<sectiontype name="email-users"
             extends="ZConfig.basic.mapping"
             keytype="mypkg.datatypes.email_address"
             />

<section name="*"
         type="email-users"
         attribute="email_users"
         />
```

5.2 ZConfig.components.logger

The `ZConfig.components.logger` package provides configuration support for the `logging` package in Python's standard library. This component can be imported using

```
<import package="ZConfig.components.logger"/>
```

This component defines two abstract types and several concrete section types. These can be imported as a unit, as above, or as three smaller components usable in creating alternate logging packages. The configuration component for Zope's `zLOG` package, starting with Zope 2.8, is an example of using this component in this way.

The first of the three type groups contains the abstract types, and can be imported using

```
<import package="ZConfig.components.logger" file="abstract.xml"/>
```

The two abstract types imported by this are:

ZConfig.logger.log
Logger objects are represented by this abstract type.

ZConfig.logger.handler
Each logger object can have one or more "handlers" associated with them. These handlers are responsible for writing logging events to some form of output stream using appropriate formatting. The output stream may be a file on a disk, a socket communicating with a server on another system, or a series of `syslog` messages. Section types which implement this type represent these handlers.

The second type group provides section types that act as factories for `logging.Logger` objects. This can be imported using

```
<import package="ZConfig.components.logger" file="loggers.xml"/>
```

The types defined in this component implement the **ZConfig.logger.log** abstract type.

The third type group provides section types that are factories for `logging.Handler` objects. This can be imported using

```
<import package="ZConfig.components.logger" file="handlers.xml"/>
```

The types defined in this component implement the **ZConfig.logger.handler** abstract type.

See Also:

PEP 282, "*A Logging System*"
The proposal which described the logging feature for inclusion in the Python standard library.

Python's `logging` package documentation

Original Python `logging` package
This is the original source for the `logging` package. The version of the package available from this site is suitable for use with Python 2.1.x and 2.2.x, which do not include the `logging` package in the standard library.

6 Using Components to Extend Schema

It is possible to use schema components and the `%import` construct to extend the set of section types available for a specific configuration file, and allow the new components to be used in place of standard components.

The key to making this work is the use of abstract section types. Wherever the original schema accepts an abstract type, it is possible to load new implementations of the abstract type and use those instead of, or in addition to, the implementations loaded by the original schema.

Abstract types are generally used to represent interfaces. Sometimes these are interfaces for factory objects, and sometimes not, but there's an interface that the new component needs to implement. What interface is required should be documented in the `description` element in the `abstracttype` element; this may be by reference to an interface specified in a Python module or described in some other bit of documentation.

The following things need to be created to make the new component usable from the configuration file:

1. An implementation of the required interface.
2. A schema component that defines a section type that contains the information needed to construct the component.
3. A "datatype" function that converts configuration data to an instance of the component.

For simplicity, let's assume that the implementation is defined by a Python class.

The example component we build here will be in the `noise` package, but any package will do. Components loadable using `%import` must be contained in the 'component.xml' file; alternate filenames may not be selected by the `%import` construct.

Create a ZConfig component that provides a section type to support your component. The new section type must declare that it implements the appropriate abstract type; it should probably look something like this:

```
<component prefix="noise.server">
  <import package="ZServer"/>

  <sectiontype name="noise-generator"
               implements="ZServer.server"
               datatype=".NoiseServerFactory">

    <!-- specific configuration data should be described here -->

    <key name="port"
         datatype="port-number"
         required="yes">
      <description>
        Port number to listen on.
      </description>
    </key>

    <key name="color"
         datatype=".noise_color"
         default="white">
      <description>
        Silly way to specify a noise generation algorithm.
      </description>
    </key>

  </sectiontype>
</component>
```

This example uses one of the standard ZConfig datatypes, **port-number**, and requires two additional types to be provided by the `noise.server` module: `NoiseServerFactory` and `noise_color()`.

The `noise_color()` function is a datatype conversion for a key, so it accepts a string and returns the value that should be used:

```
_noise_colors = {
    # color -> r,g,b
    'white': (255, 255, 255),
    'pink':  (255, 182, 193),
    }

def noise_color(string):
    if string in _noise_colors:
        return _noise_colors[string]
    else:
        raise ValueError('unknown noise color: %r' % string)
```

`NoiseServerFactory` is a little different, as it's the datatype function for a section rather than a key. The parameter isn't a string, but a section value object with two attributes, `port` and `color`.

Since the **ZServer.server** abstract type requires that the component returned is a factory object, the datatype function can be implemented at the constructor for the class of the factory object. (If the datatype function could select different implementation classes based on the configuration values, it makes more sense to use a simple function that returns the appropriate implementation.)

A class that implements this datatype might look like this:

```
from ZServer.datatypes import ServerFactory
from noise.generator import WhiteNoiseGenerator, PinkNoiseGenerator

class NoiseServerFactory(ServerFactory):

    def __init__(self, section):
        # host and ip will be initialized by ServerFactory.prepare()
        self.host = None
        self.ip = None
        self.port = section.port
        self.color = section.color

    def create(self):
        if self.color == 'white':
            generator = WhiteNoiseGenerator()
        else:
            generator = PinkNoiseGenerator()
        return NoiseServer(self.ip, self.port, generator)
```

You'll need to arrange for the package containing this component is available on Python's `sys.path` before the configuration file is loaded; this is mostly easily done by manipulating the PYTHONPATH environment variable.

Your configuration file can now include the following to load and use your new component:

```
%import noise

<noise-generator>
  port 1234
  color white
</noise-generator>
```

7 ZConfig — Basic configuration support

The main `ZConfig` package exports these convenience functions:

loadConfig(*schema, url*[*, overrides*])

Load and return a configuration from a URL or pathname given by *url*. *url* may be a URL, absolute pathname, or relative pathname. Fragment identifiers are not supported. *schema* is a reference to a schema loaded by `loadSchema()` or `loadSchemaFile()`. The return value is a tuple containing the configuration object and a composite handler that, when called with a name-to-handler mapping, calls all the handlers for the configuration.

The optional *overrides* argument represents information derived from command-line arguments. If given, it must be either a sequence of value specifiers, or `None`. A *value specifier* is a string of the form *optionpath*=*value*. The *optionpath* specifies the "full path" to the configuration setting: it can contain a sequence of names, separated by '`/`' characters. Each name before the last names a section from the configuration file, and the last name corresponds to a key within the section identified by the leading section names. If *optionpath* contains only one name, it identifies a key in the top-level schema. *value* is a string that will be treated just like a value in the configuration file.

loadConfigFile(*schema, file*[*, url*[*, overrides*]])

Load and return a configuration from an opened file object. If *url* is omitted, one will be computed based on the `name` attribute of *file*, if it exists. If no URL can be determined, all `%include` statements in the configuration must use absolute URLs. *schema* is a reference to a schema loaded by `loadSchema()` or `loadSchemaFile()`. The return value is a tuple containing the configuration object and a composite handler that, when called with a name-to-handler mapping, calls all the handlers for the configuration. The *overrides* argument is the same as for the `loadConfig()` function.

loadSchema(*url*)

Load a schema definition from the URL *url*. *url* may be a URL, absolute pathname, or relative pathname. Fragment identifiers are not supported. The resulting schema object can be passed to `loadConfig()` or `loadConfigFile()`. The schema object may be used as many times as needed.

loadSchemaFile(*file*[*, url*])

Load a schema definition from the open file object *file*. If *url* is given and not `None`, it should be the URL of resource represented by *file*. If *url* is omitted or `None`, a URL may be computed from the `name` attribute of *file*, if present. The resulting schema object can be passed to `loadConfig()` or `loadConfigFile()`. The schema object may be used as many times as needed.

The following exceptions are defined by this package:

exception ConfigurationError

Base class for exceptions specific to the `ZConfig` package. All instances provide a `message` attribute that describes the specific error, and a `url` attribute that gives the URL of the resource the error was located in, or `None`.

exception ConfigurationSyntaxError

Exception raised when a configuration source does not conform to the allowed syntax. In addition to the `message` and `url` attributes, exceptions of this type offer the `lineno` attribute, which provides the line number at which the error was detected.

exception DataConversionError

Raised when a data type conversion fails with `ValueError`. This exception is a subclass of both `ConfigurationError` and `ValueError`. The `str()` of the exception provides the explanation from the original `ValueError`, and the line number and URL of the value which provoked the error. The following additional attributes are provided:

Attribute	Value
`colno`	column number at which the value starts, or `None`
`exception`	the original `ValueError` instance
`lineno`	line number on which the value starts
`message`	`str()` returned by the original `ValueError`
`value`	original value passed to the conversion function
`url`	URL of the resource providing the value text

exception `SchemaError`
Raised when a schema contains an error. This exception type provides the attributes `url`, `lineno`, and `colno`, which provide the source URL, the line number, and the column number at which the error was detected. These attributes may be `None` in some cases.

exception `SchemaResourceError`
Raised when there's an error locating a resource required by the schema. This is derived from `SchemaError`. Instances of this exception class add the attributes `filename`, `package`, and `path`, which hold the filename searched for within the package being loaded, the name of the package, and the `__path__` attribute of the package itself (or `None` if it isn't a package or could not be imported).

exception `SubstitutionReplacementError`
Raised when the source text contains references to names which are not defined in *mapping*. The attributes `source` and `name` provide the complete source text and the name (converted to lower case) for which no replacement is defined.

exception `SubstitutionSyntaxError`
Raised when the source text contains syntactical errors.

7.1 Basic Usage

The simplest use of `ZConfig` is to load a configuration based on a schema stored in a file. This example loads a configuration file specified on the command line using a schema in the same directory as the script:

```
import os
import sys
import ZConfig

try:
    myfile = __file__
except NameError:
    # really should follow symlinks here:
    myfile = sys.argv[0]

mydir = os.path.dirname(os.path.abspath(myfile))

schema = ZConfig.loadSchema(os.path.join(mydir, 'schema.xml'))
conf, handler = ZConfig.loadConfig(schema, sys.argv[1])
```

If the schema file contained this schema:

```
<schema>
  <key name='server' required='yes'/>
  <key name='attempts' datatype='integer' default='5'/>
</schema>
```

and the file specified on the command line contained this text:

```
# sample configuration

server www.example.com
```

then the configuration object `conf` loaded above would have two attributes:

Attribute	Value
`server`	`'www.example.com'`
`attempts`	`5`

8 `ZConfig.datatypes` — Default data type registry

The `ZConfig.datatypes` module provides the implementation of the default data type registry and all the standard data types supported by `ZConfig`. A number of convenience classes are also provided to assist in the creation of additional data types.

A *datatype registry* is an object that provides conversion functions for data types. The interface for a registry is fairly simple.

A *conversion function* is any callable object that accepts a single argument and returns a suitable value, or raises an exception if the input value is not acceptable. `ValueError` is the preferred exception for disallowed inputs, but any other exception will be properly propagated.

class `Registry`([*stock*])

Implementation of a simple type registry. If given, *stock* should be a mapping which defines the "built-in" data types for the registry; if omitted or `None`, the standard set of data types is used (see section 4, "Standard `ZConfig` Datatypes").

`Registry` objects have the following methods:

`get`(*name*)

Return the type conversion routine for *name*. If the conversion function cannot be found, an (unspecified) exception is raised. If the name is not provided in the stock set of data types by this registry and has not otherwise been registered, this method uses the `search()` method to load the conversion function. This is the only method the rest of `ZConfig` requires.

`register`(*name, conversion*)

Register the data type name *name* to use the conversion function *conversion*. If *name* is already registered or provided as a stock data type, `ValueError` is raised (this includes the case when *name* was found using the `search()` method).

`search`(*name*)

This is a helper method for the default implementation of the `get()` method. If *name* is a Python dotted-name, this method loads the value for the name by dynamically importing the containing module and extracting the value of the name. The name must refer to a usable conversion function.

The following classes are provided to define conversion functions:

class `MemoizedConversion`(*conversion*)

Simple memoization for potentially expensive conversions. This conversion helper caches each successful conversion for re-use at a later time; failed conversions are not cached in any way, since it is difficult to raise a meaningful exception providing information about the specific failure.

class `RangeCheckedConversion`(*conversion*[, *min*[, *max*]])

Helper that performs range checks on the result of another conversion. Values passed to instances of this conversion are converted using *conversion* and then range checked. *min* and *max*, if given and not `None`, are the inclusive endpoints of the allowed range. Values returned by *conversion* which lay outside the range described by *min* and *max* cause `ValueError` to be raised.

class `RegularExpressionConversion`(*regex*)
Conversion that checks that the input matches the regular expression *regex*. If it matches, returns the input, otherwise raises `ValueError`.

9 `ZConfig.loader` — Resource loading support

This module provides some helper classes used by the primary APIs exported by the `ZConfig` package. These classes may be useful for some applications, especially applications that want to use a non-default data type registry.

class `Resource`(*file, url*[, *fragment*])
Object that allows an open file object and a URL to be bound together to ease handling. Instances have the attributes `file`, `url`, and `fragment` which store the constructor arguments. These objects also have a `close()` method which will call `close()` on *file*, then set the `file` attribute to `None` and the `closed` to `True`.

class `BaseLoader`()
Base class for loader objects. This should not be instantiated directly, as the `loadResource()` method must be overridden for the instance to be used via the public API.

class `ConfigLoader`(*schema*)
Loader for configuration files. Each configuration file must conform to the schema *schema*. The `load*()` methods return a tuple consisting of the configuration object and a composite handler.

class `SchemaLoader`([*registry*])
Loader that loads schema instances. All schema loaded by a `SchemaLoader` will use the same data type registry. If *registry* is provided and not `None`, it will be used, otherwise an instance of `ZConfig.datatypes.Registry` will be used.

9.1 Loader Objects

Loader objects provide a general public interface, an interface which subclasses must implement, and some utility methods.

The following methods provide the public interface:

`loadURL`(*url*)
Open and load a resource specified by the URL *url*. This method uses the `loadResource()` method to perform the actual load, and returns whatever that method returns.

`loadFile`(*file*[, *url*])
Load from an open file object, *file*. If given and not `None`, *url* should be the URL of the resource represented by *file*. If omitted or `None`, the `name` attribute of *file* is used to compute a `file:` URL, if present. This method uses the `loadResource()` method to perform the actual load, and returns whatever that method returns.

The following method must be overridden by subclasses:

`loadResource`(*resource*)
Subclasses of `BaseLoader` must implement this method to actually load the resource and return the appropriate application-level object.

The following methods can be used as utilities:

`isPath`(*s*)

Return true if *s* should be considered a filesystem path rather than a URL.

normalizeURL(*url-or-path*)
Return a URL for *url-or-path*. If *url-or-path* refers to an existing file, the corresponding `file:` URL is returned. Otherwise *url-or-path* is checked for sanity: if it does not have a schema, `ValueError` is raised, and if it does have a fragment identifier, `ConfigurationError` is raised. This uses `isPath()` to determine whether *url-or-path* is a URL of a filesystem path.

openResource(*url*)
Returns a resource object that represents the URL *url*. The URL is opened using the `urllib2.urlopen()` function, and the returned resource object is created using `createResource()`. If the URL cannot be opened, `ConfigurationError` is raised.

createResource(*file, url*)
Returns a resource object for an open file and URL, given as *file* and *url*, respectively. This may be overridden by a subclass if an alternate resource implementation is desired.

10 ZConfig.cmdline — Command-line override support

This module exports an extended version of the `ConfigLoader` class from the `ZConfig.loader` module. This provides support for overriding specific settings from the configuration file from the command line, without requiring the application to provide specific options for everything the configuration file can include.

class ExtendedConfigLoader(*schema*)
Construct a `ConfigLoader` subclass that adds support for command-line overrides.

The following additional method is provided, and is the only way to provide position information to associate with command-line parameters:

addOption(*spec*[, *pos*])
Add a single value to the list of overridden values. The *spec* argument is a value specified, as described for the `ZConfig.loadConfig()` function. A source position for the specifier may be given as *pos*. If *pos* is specified and not `None`, it must be a sequence of three values. The first is the URL of the source (or some other identifying string). The second and third are the line number and column of the setting. These position information is only used to construct a `DataConversionError` when data conversion fails.

11 ZConfig.substitution — String substitution

This module provides a basic substitution facility similar to that found in the Bourne shell (**sh** on most UNIX platforms).

The replacements supported by this module include:

Source	Replacement	Notes
`$$`	`$`	(1)
`$`*name*	The result of looking up *name*	(2)
`${`*name*`}`	The result of looking up *name*	

Notes:

(1) This is different from the Bourne shell, which uses \`$` to generate a '`$`' in the result text. This difference avoids having as many special characters in the syntax.

(2) Any character which immediately follows *name* may not be a valid character in a name.

In each case, *name* is a non-empty sequence of alphanumeric and underscore characters not starting with a digit. If there is not a replacement for *name*, the exception `SubstitutionReplacementError` is raised. Note that the lookup is expected to be case-insensitive; this module will always use a lower-case version of the name to perform the query.

This module provides these functions:

`substitute`(*s, mapping*)
: Substitute values from *mapping* into *s*. *mapping* can be a `dict` or any type that supports the `get()` method of the mapping protocol. Replacement values are copied into the result without further interpretation. Raises `SubstitutionSyntaxError` if there are malformed constructs in *s*.

`isname`(*s*)
: Returns `True` if *s* is a valid name for a substitution text, otherwise returns `False`.

11.1 Examples

```
>>> from ZConfig.substitution import substitute
>>> d = {'name': 'value',
...      'top': '$middle',
...      'middle' : 'bottom'}
>>>
>>> substitute('$name', d)
'value'
>>> substitute('$top', d)
'$middle'
```

A Schema Document Type Definition

The following is the XML Document Type Definition for `ZConfig` schema:

```
<!--
  ***********************************************************************
  Copyright (c) 2002, 2003 Zope Corporation and Contributors.
  All Rights Reserved.

  This software is subject to the provisions of the Zope Public License,
  Version 2.0 (ZPL).  A copy of the ZPL should accompany this distribution.
  THIS SOFTWARE IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED
  WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
  WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS
  FOR A PARTICULAR PURPOSE.
  ***********************************************************************

  Please note that not all documents that conform to this DTD are
  legal ZConfig schema.  The ZConfig reference manual describes many
  constraints that are important to understanding ZConfig schema.
  -->

<!-- DTD for ZConfig schema documents. -->

<!ELEMENT schema (description?, metadefault?, example?,
                  import*,
                  (sectiontype | abstracttype)*,
```

```
                  (section | key | multisection | multikey)*)>
<!ATTLIST schema
          extends    NMTOKEN  #IMPLIED
          prefix     NMTOKEN  #IMPLIED
          handler    NMTOKEN  #IMPLIED
          keytype    NMTOKEN  #IMPLIED
          datatype   NMTOKEN  #IMPLIED>

<!ELEMENT component (description?, (sectiontype | abstracttype)*)>
<!ATTLIST component
          prefix     NMTOKEN  #IMPLIED>

<!ELEMENT extension (description?, (sectiontype | abstracttype)*)>
<!ATTLIST extension
          prefix     NMTOKEN  #IMPLIED>

<!ELEMENT import EMPTY>
<!ATTLIST import
          file       CDATA    #IMPLIED
          package    NMTOKEN  #IMPLIED
          src        CDATA    #IMPLIED>

<!ELEMENT description (#PCDATA)*>
<!ATTLIST description
          format     NMTOKEN  #IMPLIED>

<!ELEMENT metadefault (#PCDATA)*>
<!ELEMENT example     (#PCDATA)*>

<!ELEMENT sectiontype (description?,
                       (section | key | multisection | multikey)*)>
<!ATTLIST sectiontype
          name       NMTOKEN  #REQUIRED
          prefix     NMTOKEN  #IMPLIED
          keytype    NMTOKEN  #IMPLIED
          datatype   NMTOKEN  #IMPLIED
          implements NMTOKEN  #IMPLIED
          extends    NMTOKEN  #IMPLIED>

<!ELEMENT abstracttype (description?)>
<!ATTLIST abstracttype
          name       NMTOKEN  #REQUIRED
          prefix     NMTOKEN  #IMPLIED>

<!ELEMENT default    (#PCDATA)*>
<!ATTLIST default
          key        CDATA    #IMPLIED>

<!ELEMENT key (description?, metadefault?, example?, default*)>
<!ATTLIST key
          name       CDATA    #REQUIRED
          attribute  NMTOKEN  #IMPLIED
          datatype   NMTOKEN  #IMPLIED
          handler    NMTOKEN  #IMPLIED
          required   (yes|no) "no"
          default    CDATA    #IMPLIED>

<!ELEMENT multikey (description?, metadefault?, example?, default*)>
<!ATTLIST multikey
```

```
          name       CDATA    #REQUIRED
          attribute  NMTOKEN  #IMPLIED
          datatype   NMTOKEN  #IMPLIED
          handler    NMTOKEN  #IMPLIED
          required   (yes|no) "no">

<!ELEMENT section (description?)>
<!ATTLIST section
          name       CDATA    #REQUIRED
          attribute  NMTOKEN  #IMPLIED
          type       NMTOKEN  #REQUIRED
          handler    NMTOKEN  #IMPLIED
          required   (yes|no) "no">

<!ELEMENT multisection (description?)>
<!ATTLIST multisection
          name       CDATA    #REQUIRED
          attribute  NMTOKEN  #IMPLIED
          type       NMTOKEN  #REQUIRED
          handler    NMTOKEN  #IMPLIED
          required   (yes|no) "no">
```