itools

Juan David Ibáñez Palomar[1]
jdavid@itaapy.com

March 21, 2006

Contents

Chapter 1

Introduction

1.1 This is Free Software

The software package `itools` is released to the world under the terms and conditions of the GNU General Public License[1], developers own the copyright of the code they write, contributions are very welcomed.

1.2 What is `itools`?

`itools` is a Python[2] library, it groups a number of packages into a single meta-package for easier development and deployment. The packages included are:

```
itools.catalog     itools.i18n        itools.web
itools.cms         itools.ical        itools.workflow
itools.csv         itools.resources   itools.xhtml
itools.datatypes   itools.rss         itools.xliff
itools.gettext     itools.schemas     itools.xml
itools.handlers    itools.tmx
itools.html        itools.uri
```

The Figure 1.1 shows the dependencies between the different packages. Below a brief description of each package (sorted by alphabetical order):

- `itools.catalog` – An index and search engine. It provides full text indexing as well as keyword indexing; and boolean, phrase and range search.
- `itools.cms` – A web application framework. The acronym `cms` stands for *Content Management System*. This is the top package, which uses almost everything else in `itools`.
- `itools.csv` – High level support for CSV files.
- `itools.datatypes` – Type marshalers for basic types (integer, date, etc.) and not so basic types (filenames, XML qualified names, etc.).

[1] `http://www.gnu.org/copyleft/gpl.html`
[2] `http://www.python.org`

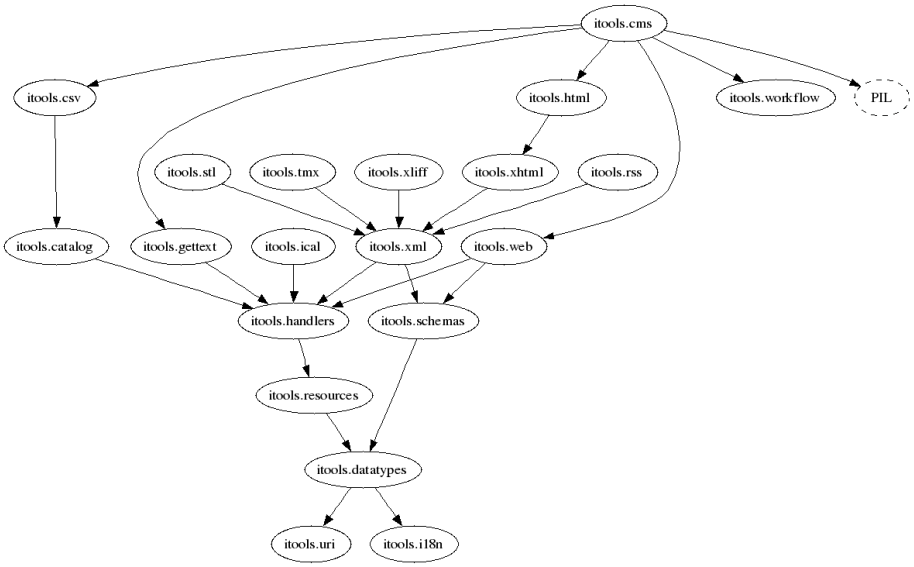

Figure 1.1: Dependency diagram

- `itools.gettext` – Support for the PO and MO file formats of the GNU gettext tools.
- `itools.handlers` – Resource handlers are non persistent classes responsible to manage a resource (a file or a folder), they represent a resource as a data structure on memory and provide an API to work with the resource.

  Included are the resource handlers for folders, binary and text files, images, CSV, etc.
- `itools.html` – Support for HTML documents.
- `itools.i18n` – Miscellaneous tools for internationalization and localization (language negotiation, text segmentation, fuzzy matching, etc.).
- `itools.ical` – Support for the iCalendar standard (RFC 2445).
- `itools.resources` – An abstraction layer over resources (files and folders), provides a the same programming interdace accross different storages.
- `itools.rss` – Support for RSS 2.0.
- `itools.schemas` – Infrastructure for datatypes schemas. Support for Dublin Core included.
- `itools.stl` – The **S**imple **T**emplate **L**anguage, a templating language for XML documents.
- `itools.tmx` – Support for the industry standard *Translation Memory eXchange*.
- `itools.uri` – Support for *Uniform Resource Indentifiers* (RFC 2396).

- `itools.web` – Abstract programming interface for web applications. Proof-of-concept implementations for the CGI protocol and an standalone server.
- `itools.workflow` – Programming interface to implement workflow in applications.

  Workflows are represented as automatons, objects can move from one state to another through transitions, classes can add specific semantics to states and transitions.
- `itools.xhtml` – Support for XHTML documents.
- `itools.xliff` – Support for the industry standard *XML Localisation Interchange File Format*.
- `itools.xml` – Support for XML (*eXtensible Markup Language*).

  Includes an intuitive event driven parser and a DOM-like representation of XML documents.

1.3 Highlights

1.3.1 The resource-handler model

Of everything in `itools` I am probably most proud of the the three sub-packages `itools.uri`, `itools.resources` and `itools.handlers`, which make up what I call the *resource-handler* model.

There is a linear relationship of dependency between these modules, `uri` depends on nothing but Python, `resources` depends on `uri`, and `handlers` depends on `resources`.

This allows to use these modules on a flexible way. For example, you can just use `itools.uri` (ignoring the others) to benefit from a higher level API to work with URIs than that provided by the Standard Library (`urlparse`).

Or you may use `itools.resources` if you want to benefit from an abstraction layer over the storage. This is to say, if you want to be able to manipulate many resources stored in different systems and accessed through different protocols with the same, consistent, rich API.

You may even take advantage of `itools.handlers` without fully understanding the model behind. For example, you could use some of the handlers `itools` offers out-of-the-box for standard file formats like CSV, PO or XHTML, to simplify your life if you ever need to work with one of these formats.

But, in order to exploit everything `itools` has to offer to its limit, you may choose to base part or all of your application architecture on the *resource-handler* model, a model heavily influenced by the filesystem and the Web. In this situation the three packages (`uri`, `resources` and `handlers`) show themselves as three different components with a distinct role in the architecture:

- `itools.uri` identifies and locates resources, wherever they are. Altogether with `itools.resources` it effectively enables your information system to be distributed through an heterogeneous base.
- `itools.resources` provides persistency to your data, this is to say, resources are where the data is stored.

- and `itools.handlers` is where the logic lives. The essential characteristic of a resource handler (or handler for short) is that it is non-persistent, instead it is associated with a resource, whose content it is responsible to manage.

The first chapters of this document will cover these sub-packages, `itools.uri`, `itools.resources` and `itools.handlers` with detail.

1.3.2 eXtensible Markup Language

The `itools.xml` package depends on `itools.handlers`, this is its first advantage. It represents an XML document as an DOM like tree. But it also provides support for schemas, what allows to easily build higher level data structures.

The package also provides handlers for specific document types out-of-the-box, most notably XHTML and HTML (even if HTML is not XML, it shares a lot with XHTML).

1.3.3 The Simple Template Language

The Simple Template Language is included in the `itools.xml` package, but it is important enough to deserve its own chapter in the documentation.

Its design goals are:

- Truly separate logic and presentation. Even Python expressions are not allowed within the template.
- Really simple. It can be mastered in half a day.
- Damn fast (easy to achieve through simplicity).
- Secure. Even non-trusted users could write templates without risk, because code is not allowed within the template.

And the key idea behind, is to make it a *descriptive* language.

1.3.4 Workflow

The sub-package `itools.workflow` (once known as *flux*) is the oldest code in `itools`. It does not depend on anything but the Standard Library, and don't puts any restriction on the storage. This makes it very easy to combine with other frameworks.

There is a chapter exclusively dedicated to it.

1.3.5 Internationalization and Localization

The `itools.i18n` package provides a wide range of tools for internationalization and localization of both software and data. From message extraction to language negotiation, through text segmentation, fuzzy matching or algorithms to guess the language a text is written in.

A chapter is devoted to this topic.

1.3.6 Index and Search

The `itools.catalog` sub-package provides an index and search engine. Though still young, it already provides full text indexing, boolean queries and results sorted by weight.

1.4 Project status

Currently `itools` is under active development. At the time of this writing the last version available is **0.12**, what it provides is the scope of this document. Here I want to offer a glance about what is comming next:

- SQL. Relying on third party products, we will explore relational databases and see what we can do to simplify its use. Specifically the goal is to provide a programming interface that seemlessly integrates with the rest of `itools`.
- Standards support. `itools` is strongly focused on the implemetation of standards. While we will add new standards to the collection, in the short-term the focus will be on improving the support we already provide for some of them, just two spotlights:
  - A native parser for XML would increase the control we have of the parsing process, to allow things like parsing fragments. Also, some rough corners of the XML API must be polished.
  - CSV is a very common format, hence to have a very good programming interface is more of a priority. Probably a new package, `itools.csv`, will be implemented.
- Performance. While performance is already one of the strong points of `itools`, as the *Simple Template Language* illustrates, it is not enough:
  - One thing we need is better benchmarks, so we can better measure how `itools` compares to other alternatives, specially for the catalog.
  - Support for lazy load will be added to the most important handler classes.
  - While profiling the CPU usage is easy with the Python profiler, the lack of similar tools to measure the memory consumption makes it hard to optimize this aspect. Anyway some infrastructure must be put in place.
- Documentation, is critical. The target for this document is to cover all of `itools`. Also some changes in the structure may arise, one idea is have one complete reference guide, plus a collection of tutorials.
- Lower the entry point, improve the development process. Today we use *tla 1* to manage the sources, it is great when you master it, but is hard to get there. It looks like the new version, `tla 2`, is addressing this problem; other alternatives like bazaar are worth considering.

  To keep improving the development process of `itools` is a constant we don't stop working on.

For further information you are welcome to the `itools` mailing list; the bug-tracker is available to report bugs and suggestions; the sources can be browsed through the web; and don't forget to check the web site for last downloads and news:

- Mailing list, `http://in-girum.net/mailman/listinfo/ikaaro`
- Bug Tracker, `http://bugs.ikaaro.org`
- Browse the sources, `http://lleu.org/cgi-bin/gitweb.cgi`
- The web site, `http://www.ikaaro.org/itools`

1.5 Installation

`itools` requires Python 2.4, earlier versions are not supported.

The last version of `itools` can be downloaded from `http://www.ikaaro.org`. It is distributed as tarball, download and unpack it somewhere; then install with *distutils*, type:

```
$ python setup.py install
```

Be sure to have the right permissions.

1.6 About this document

This paper is the official documentation of `itools`. It is addressed to Python developers. Though it may be useful to software architects in general, as some ideas exposed here may be found interesting[3].

It is convenient to have some basic skills with the Python programming language to fully understand this document. Probably the best introduction to Python is the official tutorial:

`http://python.org/doc/2.4.1/tut/tut.html`.

This document touches many different technologies and standards, such as XML. References will be given in the relevant chapters.

Currently the document covers around 60% of the `itools` package. There are also two appendixes, one explaining the coding style `itools` is written in, another one introducing the use of *GNU arch*. Both are specially addressed to those that want to contribute back to the main tree.

[3]For example, `itools.workflow` inspired XXX to write a similar engine in Java, see `http://XXX`

Chapter 2

Catalog

The (`itools.catalog`) package provides an index and search engine. There are not many solutions of its kind available to Python developers, the most popular are:

- Lupy[1], an implementation in Python of the Lucene[2] engine. The main drawback is that it is not maintained anymore.
- PyLucene[3], a wrapper around Lucene that allows to use it directly from Python. It is somewhat complex to deploy.
- ZCatalog[4], the solution for Zope 2[5]. It only works with the ZODB[6] (unable to index a file in the filesystem or a web page for example), and its future is uncertain with the new Zope 3[7].

In this context `itools.catalog` offers a nice alternative. Fully written in Python, it is easy to use and deploy; able to index anything; actively maintained and fast enough for small to medium applications.

2.1 Quick Start

To illustrate the usage of `itools.catalog`, we are going to index and search the Web! I mean, a couple of pages.

1. Create a catalog object (in memory) with just two fields:

```
>>> from itools.catalog.Catalog import Catalog
>>> catalog = Catalog(fields=[('url', 'keyword', True, True)])
...                           ('body', 'text', True, False)])
```

2. Index a couple of pages:

[1] http://XXX
[2] http://XXX
[3] http://XXX
[4] XXX
[5] XXX
[6] XXX
[7] XXX

```
>>> from itools.handlers import get_handler
>>> import itools.html
>>>
>>> for url in ['http://www.slashdot.org', 'http://www.XXX']:
...     page = get_handler(url)
...     document = {'url': url, 'body': page.to_text()}
...     catalog.index_document(document)
```

3. Search now:

```
>>> for document in catalog.search(body='python'):
...     print document.url
XXX
```

2.2 Architecture

It is good to have the *big picture* in the head when working with an index and search engine, and it is pretty simple actually. A catalog is made up of four things:

reverse indexes The internal data structure used to perform the searches. In our example there are two reversed indexes, one for the url of the web page, the other for its text body.

stored fields This is actually a cache, it keeps the original values for quick access. In our example only the title is stored (we don't store the body to keep the size of the catalog small).

internal document identifier When a document is indexed it is assigned a *unique id* (a number). From the point of view of the public API the internal *unique id* is only used when unindexing an object.

external document identifier Actually, the Catalog knows nothing about an *external id*; but you must, because the *external id* will be the way you will reach the original object after a search. In our example the *external id* is the URL. The *external id* must be both indexed and stored.

Operations

The operations that can be done with a catalog are:

Index/Unindex As well as indexing an object we may want to remove it from the catalog (unindex). To update a document first it must be unindexed, and then indexed again.

Search Of course the primary goal to index something is to be able to search it later. The result of a search operation will be a sequence of objects; not the original documents that were indexed, but a special object that offers the stored fields for quick access, and also the *unique id* (`__number__`).

2.3 Indexing

2.3.1 The Catalog object

The fields that are indexed and stored are defined at creation time. The `Catalog` object expects as parameter a list where each element is a four-elements tuple defining a field to be indexed or stored, or both. The four elements are:

field name The name of the field.

field type An string defining the type of the field; the way a field is indexed depends on its type. Possible values are: `text`, `bool`, `keyword` and `path`.

indexed A boolean value that specifies wether the field will be indexed or not (to be able to search for a field it must be indexed).

stored A boolen value that specifies wether the field will be stored or not. The value of an stored field can be quickly retrieved on search.

In our example we have two fields, the *url* and the *body*. The first is a *keyword* field, the last is a *text* field. Both are indexed, only the *url* is stored:

```
>>> catalog = Catalog(fields=[('url', 'keyword', True, True)])
...                           ('body', 'text', True, False)])
```

2.3.2 Analysers

What the type of a field defines is the analyser that will be used to process the field value for indexing.

What an analyser expects as input is the field value, what it returns is a sequence of two-element tuples, where the first element is a "word" and the second is the position of that word within the original value. For example, the `Text` analyser works this way:

```
>>> from itools.catalog.Analysers import Text
>>> value = u"Indexing and searching, that's a point."
>>> for word, position in Text(value):
...     print position, word
...
0 indexing
1 and
2 searching
3 that
4 s
5 a
6 point
```

There are four analysers included in `itools.catalog.Analysers` (you may define your own):

- The most powerful is the `Text` analyser, which splits a text into words.
- Another very important is the `Keyword` analyser, because it will be the one that you will probably use for the *external id*. It is also the most simple, since it interprets the value literally.

- The `Bool` analyser is like the keyword one, except that it expects a boolean value.
- Finally there is the `Path` analyser, which expects a path as value, and will split the path into its segments (uses `itools.uri`).

The analysers are also important because you may want to use them when searching, to process the user entered search criterias.

2.3.3 Index/Unindex

The index operation, we have already seen it. Implemented by the `index_document` method, it expects a mapping from field name to field value (which is the source of the data to be indexed, the catalog does not cares about). Lets review the example code:

```
>>> for url in ['http://www.slashdot.org', 'http://www.XXX']:
...     page = get_handler(url)
...     document = {'url': url, 'body': page.to_text()}
...     catalog.index_document(document)
```

The unindexing operation is slighly more complex:

```
>>> for document in catalog.search(url='http://www.slashdot.org'):
...     catalog.unindex_document(document.__number__)
```

To update an index first remove it from the catalog with `unindex_document` and then add it again to the catalog with `index_document`.

2.4 Searching

2.4.1 The `search` method (introduction to queries)

The `search` method can be called two ways, with a query object as the only parameter, or with one or more keyword parameters (as we have seen so far).

The example used before:

```
>>> catalog.search(body='python')
```

May also be written:

```
>>> from itools.catalog import Query
>>>
>>> query = Query.Equal('body', 'python')
>>> catalog.search(query)
```

The module `itools.catalog.Query` provides a number of classes to perform basic and advanced searches. The most simple is the `Equal` query, that simple checks for fields that match a value.

The use of keyword parameters is a shorthand to perform a boolean search with the *and* operator. The hypothetical example:

```
>>> catalog.search(state='public', body='python')
```

Would return only the public documents that talk about Python (here of course we need another field, the workflow state).

2.4.2 Boolean queries

But sometimes we want to search using the *or* operator, to find out objects that match either one criteria or another. Or we may even want to create more complex boolean searches. An example explains itself:

```
>>> python = Query.Equal('body', 'python')
>>> perl = Query.Equal('body', 'perl')
>>> ruby = Query.Equal('body', 'ruby')
>>> public = Query.Equal('state', 'public')
>>>
>>> python_or_perl_or_ruby = Query.Or(python, perl, ruby)
>>> query = Query.And(public, python_or_perl_or_ruby)
>>>
>>> for document in catalog.search(query):
...
```

The boolean queries (`And`, `Or`) accept *n* sub-queries as parameters, and will search for documents that either match all the sub-queries (`And`) or at least one of them (`Or`).

2.4.3 Phrase searches

To build a query to search phrases is straightforward:

```
>>> query = Query.Phrase('body', 'Python is better than Perl')
```

2.4.4 Range searches

Now imagine we want to search for documents that have been modified since the last week (with a field *mtime*):

```
>>> from datetime import date, timedelta
>>>
>>> today = datetime.today()
>>> last_week = today - timedelta(7)
>>>
>>> today = today.strftime('%Y-%M-%d')
>>> last_week = last_week.strftime('%Y-%M-%d')
>>> query = Query.Range('mtime', last_week, today)
```

Since we don't have an specific analyser for dates we will just index the *mtime* field as a *keyword*, with the format `YYYY-MM-DD`.

It is possible to do range searches not only for dates, but for whatever values where an order relationship can be stablished.

2.5 API

2.5.1 The Catalog

- `Catalog(fields=[])`

Creates a new catalog object (in memory). The parameter `fields` is a list of tuples, each tuple defining a field that must be indexed, stored or both.

The tuple has four elements: the field name, the field type (either `'text'`, `'keyword'`, `'bool'` or `'path'`), wether the field is indexed or not (a boolean value) and wether the field is stored or not (another boolean value).

Example:

```
>>> Catalog(fields=[('url', 'keyword', True, True)])
                    ('body', 'text', True, False)])
```

- `Catalog(resource)`

  Loads a catalog object from a resource (see `itools.handlers`).

- `Catalog.index_document(document)`

  Where `document` is a mapping from field name to field value. The method returns the *internal document identifier*.

- `Catalog.unindex_document(document_id)`

  Removes from the catalog the document identified by `document_id`, the *internal document identifier*.

- `Catalog.search(query)`

  Returns a sequence with all the documents that match the given query. The sequence is ordered by weight. Each element of the sequence provides access to the stored fields, and to the *internal document identifier* (`__number__`). This method is a generator.

  Example:

  ```
  >>> for document in catalog.search(query):
  ...     print document.__number__
  ...     print document.url
  ```

- `Catalog.search(**kw)`

  Builds a boolean query from the given keyword parameters. It uses the *and* operator: it returns the documents that match *all* the given search criterias. This is a shorthand for a commonly used search.

2.5.2 Queries

- `Query.Equal(name, value)`

  Builds a simple query that will match documents whose field `name` contains the at least once the given `value`.

- `Query.Phrase(name, value)`

  Builds a phrase query that will match documents whose field `name` contains at least once all the words in `value`, in the same relative position.

- `Query.Range(name, start, end)`

  Builds a range query that will match documents whose field `name` contains a word greater than the value `start` and smaller than `end`.

- `Query.And(*args)`

  Builds a boolean query that will match documents that match *all* of the given subqueries.

- `Query.Or(*args)`

  Builds a boolean query that will match documents that match *any* of the given subqueries.

Chapter 3

CMS (a.k.a. *i*Kaaro)

The package `itools.cms` is also known as *i*Kaaro. But, what is it?

A Content Management System

It is an application, specifically, a *Content Management System*. It is very easy to deploy with the `icms` scripts (we will see them later).

And it is very simple too; its funcitonal scope, what it is able to do out-of-the-box, is quite narrow. But, don't underestimate its power, what you see is only the tip of an iceberg.

A Web Application Framework

Altogether with the rest of `itools`, *i*Kaaro is a web application framework. Personalization and extension is achieved through programming. It is the purpose of this chapter to uncover its power.

To be honest, what makes it different is not its "power", you can do with `itools.cms` as much as you can do with any other web framework. What makes it different is how low it keeps costs of building a web application, even when it scales up.

3.1 Requirements

So far `itools.cms` has been tested only in Unix environments (GNU/Linux, MacOS X, etc.), probably it won't work on the Windows platform. If you are interested in running `itools.cms` on Windows please report any problems you encounter to our bug tracker, `http://bugs.lleu.org`.

The command line tools `rsync` and `tidy` are required to run `itools.cms`. To have support for indexing and preview of office documents (PDF, OpenOffice, etc.) some other commands must be installed: `xlhtml`, `pdftotext`, `catdoc`, `ppthtml`, `iconv`, `links`, `unzip`, `pdftohtml` and `wvHtml`. Also, if the Python package `PIL`[1] is installed, it will be used to draw thumbnails.

That's all what you need. At least for testing and developing. In production it is highly recommended to have the Apache[2] web server, but we will talk about

[1] http://XXX
[2] http://http.apache.org

that later.

3.2 Deploy and manage with `icms`

When you install `itools`, you install the package `itools.cms` and also a collection of scripts:

`icms-init`	to create a new instance of itools.cms
`icms-start`	to start a web server (publishes the instance)
`icms-stop`	to stop the web server
`icms-update`	to update the instance, after a software upgrade
`icms-restore`	to recover the instance (in case of a crash)

All the scripts provide on-line documentation, just run any of them with the option `--help`, for example:

```
$ icms-init --help
usage: icms-init [OPTIONS] TARGET

Creates a new instance of itools.cms with the name TARGET.

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -p PORT, --port=PORT  listen to port number
  -r ROOT, --root=ROOT  use the ROOT handler class to init the instance
  -s SOURCE, --source=SOURCE
                        use the SOURCE directory to init the instance
  -w PASSWORD, --password=PASSWORD
                        use the given PASSWORD for the admin user
```

Now, let's see the five scripts in detail.

`icms-init`

This is the command to start with, it will create a new instance of `itools.cms` everytime you type it. This is the way I use it:

```
$ cd ikaaro-servers
$ icms-init my_instance
*
* Welcome to itools.cms
* A user with administration rights has been created for you:
*   username: admin
*   password: VLHiUgKr
*
* To start the new instance type:
*   icms-start my_instance
*
$ ls -1 my_instance
config.ini
database
database.bak
state
```

Several things to note about the code above:

- First that I like to put all my instances in a dedicated folder, `ikaaro-servers` in the example. Of course, you can create it wherever you want.
- Second, the script creates a user called `admin` with administrative rights, and a random password is generated. You will use this user to login for the first time and configure your *i*Kaaro instance.

  A good idea is to change the password to something you can remember.
- Third, a directory named `my_instance` is created. It contains a configuration file and the database. Later on, when you start working with your new instance other files will be created, most notably the log files.

Some of the options that the `icms-init` command accepts are `--password`, `--port` and `--root`.

icms-start/icms-stop

Time to run our new shiny instance:

```
$ icms-start my_instance
Listen port 8080
$ ls -1 my_instance
access_log
config.ini
database
database.bak
error_log
pid
state
```

This runs a web server (`itools.web.server`) listening at the port 8080. So, what we do?, we start our favorite web browser[3] and we open the URL `http://localhost:8080`. Figure 3.1 shows what our amazed eyes will see.

To stop the server we type:

```
$ icms-stop my_instance
Stopped.
```

If we want to run the server without detaching from the console (in debug mode) we type:

```
$ icms-start --debug my_instance
Listen port 8080
```

If we want the server to listen on another port than 8080, we can either modify the configuration file (we will see it later) or use the `--port` option:

```
$ icms-start --port 6666 my_instance
Listen port 6666
```

The command `icms-stop` does not accepts any option.

[3] Hopefully it will be a Gecko browser, like Firefox (http://www.mozilla.org).

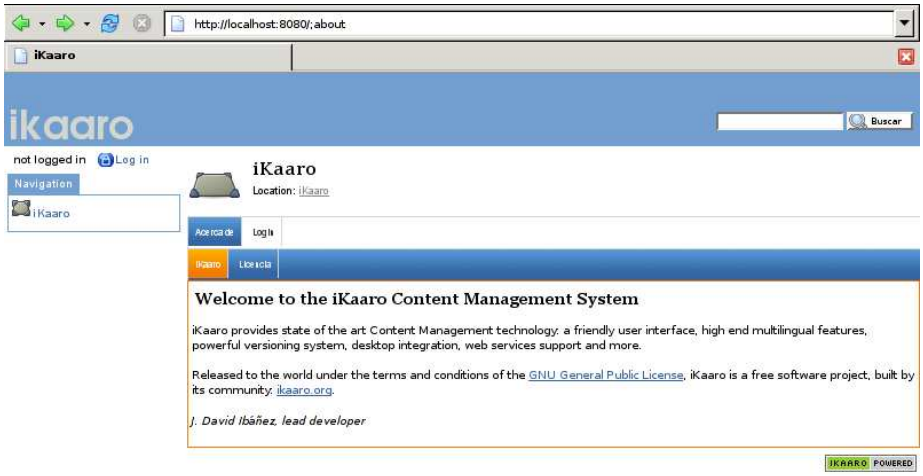

Figure 3.1: The *i*Kaaro back-office.

`icms-update`

Sometimes there is a new version of `itools.cms` that changes the layout or the format of the information it stores. When this happens the update command must be run:

```
$ icms-stop my_instance
Stopped.
$ icms-update my_instance
The instance is up-to-date (version: 20060205).
$ icms-start my_instance
Listen port 8080
```

In this example the data is up-to-date, so the command does nothing. It is highly advised to run `icms-update` with the server stopped.

`icms-restore`

Now imagine that your server is running fine, but then there is a failure in the power supply... If the crash happened in the midlle of a transaction (a write operation in the database), you may find you can not start again the server:

```
$ icms-start my_instance
The database is not in a consistent state, to fix it up type:

    $ icms-restore <instance>
```

Don't worry, our little server is crash-proof. Just do as the script suggests and run the command `icms-restore`:

```
$ icms-restore my_instance
Restore database from backup (y/N)? y
   * Restoring... DONE
```

3.2.1 The configuration file

The configuration file within the instance, called `config.ini`, follows the format of the INI files that can be found in MS Windows.

It has a single section, `instance`, which may contain two parameters:

- `port`

  The number of the port the server will listen to. By default it is `8080`.

- `modules`

  A blank-separated list of Python package names to be imported on start-up time; its purpose is to load extensions to `itools.cms`. By default the list is empty.

An example using both parameters would look like:

```
[instance]
port = 9080
modules = portal1
```

3.3 A tour through *i*Kaaro

XXX

3.4 The URL interface

Even if most users don't type the URL directly in the navigation bar, the URL remains the first and most important user interface. An `itools.cms` URL looks like this:

```
http://localhost:8080/users/admin/;profile
```

Lets highlight the path component:

```
/users/admin/;profile
```

One thing that makes this URL look different is the use of the semicolon character, which has an especial meaning[4]. Actually the path is divided in two parts:

The handler path The path from the root of the handler tree to the a resource handler we want to reach (in our example it is `/users/admin`).

The handler method The method (a view or an action) that we want to call on the resource handler (in our example it is `profile`).

The use of the semicolon maybe controversial, but it has a very interesting property, it makes URLs unambiguous.

[4]Note that the semicolon makes part of the URI standard (RFC 2396), specifically it separates the segment name from the segment parameters.

Software versus Data

We have talked about *resource handlers*, but we have not explained what they are. Resources are documented in Chapter 11 and resource handlers are documented in Chapter 7.

Here it will be enough to say that an `itools.cms` application is organized as a tree of resource handlers, and that it is these handlers are which define the semantics of our application.

XXX

3.5 Inspecting the data

With `itools.cms` the data is stored in the filesystem, within the directory `database`:

```
$ ls -1a my_instance/database
.
..
admins
.admins.metadata
.archive
.catalog
en.po
.en.po.metadata
.metadata
reviewers
.reviewers.metadata
users
.users
.users.metadata
```

This is extremely useful for instrospection and manipulation purposes, since we can use the old good Unix tools: `grep`, `vi`, etc. But of course, *don't make any changes unless you know what you are doing!*

Special resources start by a dot

Some resources are hidden and not appear (explicitly) in the back-office, these are the *special resources*, and they start by a dot: `.admins.metadata`, `.archive`, `.catalog`, `.metadata`, etc.

These special resources contain important information. A summary follows:

.*.metadata These resources contain information that describes another resource. They're important enough to be explained with more detail (see below).

.archive Where the versioning info is stored.

.catalog Where the indexes are stored.

.users All user groups contain this resource, it lists the names of the users that belong to the group. Though, for the root group, the file is empty, as all the users that belong to the root are all the users that exist in the `users` folder.

Metadata

The metadata is a central concept in *i*Kaaro. Every resource within *i*Kaaro, both files and folders (except those that start by a dot), have a metadata file that describes them, which is at the same level than the resource, and whose name is `.<resource name>.metadata`. For example, if the name of the resource is `home.html`, the name of the metadata resource is `.home.html.metadata`.

There is one exception though, the metadata resource that describes the root, it is named `.metadata`; and is within the described resource, not at the same level.

Metadata resources are XML files. We will see exactly what information they provide, and how to work with them, later in another chapter.

Reserved names

As maybe you have already guessed, resource names that start by a dot are reserved. This means that you can not add a resource whose name starts by a dot.

3.6 Extending and customizing ikaaro

The central point of `itools.cms` is not to use it as it is, but to build applications based on it, that extend its functional scope and adapt to the specific requirements of whatever web application.

And this is done through programming.

The configurability of `itools.cms` is limited, on purpuse. All the back-office interfaces are addressed to end users, and they are as complex as they should be for an end user.

The programmer will modify the *i*Kaaro behaviour to build a custom application with standard tools: a text editor like *emacs* or *vi* to write the code; a source code management tool like *git* or *tla* to maintain the source, etc.

The explanation on how to build your own application will follow a *top-down* approach, explaining how to modify different aspects of `itools.cms`, from the most basic to the most advanced.

3.6.1 Bootstrap: Portal 0

Here we explain how to build the skeleton of an application: it will do nothing beyoid a vanilla *i*Kaaro instance, but will serve as the foundation for our application.

The example code can be found in the `examples/portal0` directory, the content of our naked application will be:

```
portal0
|-- __init__.py
|-- root.py
|-- setup.py
```

A Python package

First, the portal is a Python package, hence it must be built, distributed and installed as such, with the standard *distutils* tools.

This means that we need two files, `__init__.py` and `setup.py`. The first makes our directory into a Python package, even if empty. The last is required by `distutils`.

It is not our purpose of to explain how *distutils* work, for that refer to the Python's documentation. But it is worth to see a fragment of our example's `setup.py` file:

```
# Import from itools
from itools import build_py_fixed

setup(name = "portal0",
      version = "0.1",
      ...
      cmdclass={'build_py': build_py_fixed})
```

We can not use `distutils.command.build_py.build_py` because of a bug[5] in `distutils`, instead we have to use `itools.build_py_fixed`.

The root

An `itools.cms` application is organized as a tree of *resource handlers*. The first thing we will want to do is to take control of the root of this tree.

To do so we will define a handler class named `Root` in the file `root.py`:

```
# Import from itools
from itools.cms.Root import Root as ikaaroRoot

class Root(ikaaroRoot):

    class_id = 'portal'
    class_title = u'Portal 0'
    class_description = u'A portal to learn iKaaro'

ikaaroRoot.register_handler_class(Root)
```

Several things to remark from this code:

- The handler class inherits from the *i*Kaaaro's root handler class, so we inherit its behaviour.
- We define several class variables: `class_id` is mandatory, and must be a unique identifier for our handler class; `class_title` and `class_description` are optional but highly recommended.

  There are other class variables like `class_icon16` and `class_icon48`, we will see all them later.
- We register our handler class.

[5] http://XXX

There is one last detail for our portal to work. The root handler class must be imported in the `__init__.py`:

```
from root import Root
```

Testing

To check that what we have done so far works, start installing the portal:

```
$ python setup.py build install
```

Now run the Python interpreter and check everything is fine:

```
>>> from portal0 import Root
>>> print Root.class_id
portal
```

Deployment

Ok, so the software is installed. Now we are going to build an instance for our portal:

```
$ cd ikaaro-servers/
$ icms-init --root=portal0 my_portal
*
* Welcome to itools.cms
* A user with administration rights has been created for you:
*   username: admin
*   password: TCr0oqmr
*
* To start the new instance type:
*   icms-start my_portal
*
```

What makes this process different from the process of creating a vanilla `itools.cms` instance is that we have used the `--root` parameter, whose value must be the name of a Python package following the conventions we have described before.

If everything went fine you will be able to run the instance with `icms-start`, exactly the same as with a vanilla `itools.cms` instance. Then if you go to the web browser you will be able to verify that our *Portal 0* is undistinguishable from a vanilla `itools.cms` instance.

Now we are going to start start adding features to our portal, to customize its appearance, etc.

3.6.2 Customize the graphic design (skins): Portal 1

3.6.3 Defining views: Portal 2

Chapter 4

CSV

In this chapter you will learn how to work with CSV files using the `itools.csv` handler. The explanation will be driven by an example.

4.1 What we want to do

Let's assume that you are the owner of a small ISP (Internet Service Provider) company. You have about 250 clients and all information about your clients is stored in a file named `clients.csv`.

In this file you have the client's personal data as name, surname and address and his business data: when the last payment was, how many computers the client connects to the network and when the client was registered. In the file there is also the client's discount rate. Its value depends on the number of computers, the registration date and it is change after payment.

You need an small system to store/change/remove the clients information and to generate some reports (for example to send the payment email remainders to the clients). We will build that system from scratch with `itools.csv`.

Take a closer look at the structure of the `clients.csv` file. The column names (defined in handler schema, more info about it later) and its type. The types are defined in `itools.datatypes` module.

Column name	Data type	Description
client_id	Integer	client ID
surname	Unicode	client's surname
name	Unicode	client's name
address	Unicode	client's address
email	Unicode	clients's email address
last_pay_date	Date	the date of the last pay
num_of_computers	Integer	how many computers are connected
register_date	Date	the registration date
discount	Integer	discount rate

The system should allow us to:

- get the list of clients
- add a new clients

- remove clients
- modify the information about clients
- get info about sending payment reminders

4.2 Building the system

How to do this task with `itools.csv`? To work with the database we should create the csv file handler:

```
>>> from itools.csv import CSV
>>> from itools.resources import get_resource
>>> resource = get_resource('clients.csv')
>>> clients = CSV(resource)
```

How can we view the client from the first file row?

```
>>> clients.get_row(0)
[u'1', u'Piotr', u'Macuk', u'Starowiejska 25/2 81-465 Gdynia',
u'piotr@macuk.pl', u'2004-11-30', u'2', u'2001-01-05', u'35']
```

We can add a new client to the `clients.csv` database:

```
>>> clients.add_row([4, 'Hanna', 'Nowak', 'Dlugi Targ 32 80-112 Gdansk',
'hanka@onet.pl', '2005-11-12', 1, '2005-11-12', 0])
```

To remove the client from the third row (indexes start with 0) lest's try:

```
>>> clients.del_row(2)
```

As you can realize to do the above tasks, we should know the row index which the client's data are stored in the `clients.csv` file database. We have the *client_id*, but the csv handler has no idea about that. We should inform the handler about the structure of the database. To do that we define the columns and the schema of our clients' data and load the file content again. The csv file schema is dictionary mapping from column name to datatype. We can also define columns to index. Indexed columns can be used to search data by the column values.

```
>>> from itools.datatypes import Integer, Unicode, Date
>>> class MyClients(CSV):
...     columns = ['client_id', 'surname', 'name', 'address', 'email',
...                'last_pay_date', 'num_of_computers', 'register_date',
...                'discount']
...     schema = {'client_id': Integer(index=True),
...               'surname': Unicode(index=True),
...               'name': Unicode, 'address': Unicode,
...               'email': Unicode(index=True),
...               'last_pay_date': Date(index=True),
...               'num_of_computers': Integer,
...               'register_date': Date, 'discount': Integer}
>>> my_clients = MyClients(resource)
```

And now when we get the first row from our database, the data will have the appropriate types (the above get_row(0) function call returned data as unicode strings).

```
>>> my_clients.get_row(0)
[1, u'Piotr', u'Macuk', u'Starowiejska 25/2 81-465 Gdynia',
u'piotr@macuk.pl', datetime.date(2004, 11, 30), 2,
datetime.date(2001, 1, 5), 35]
```

The index parameter (for example: `'client_id': Integer(index=True)`) will tell the handler to index that column. We want to index *client_id*, *surname*, *email* and *last_pay_date*.

Now, when we have the indexed *client_id* column we can get the client by the ID. We should find the row index with the appropriate *client_id*, and get that row from `clients.csv` file.

```
>>> indexes = my_clients.search(client_id=1)
>>> my_clients.get_row(indexes[0])
[1, u'Piotr', u'Macuk', u'Starowiejska 25/2 81-465 Gdynia',
u'piotr@macuk.pl', datetime.date(2004, 11, 30), 2,
datetime.date(2001, 1, 5), 35]
```

The search method returns the list of rows which include `client_id = 1`. That column is our unique client ID number thus we have to get the first element from that list (`indexes[0]`).

How we can find clients to send the payment reminder? Let's try:

```
>>> from itools.catalog import Query
>>> query = Query.LessOrEqual('last_pay_date',
...                           Date.decode('2004-11-30'))
>>> indexes = my_clients.search(query)
>>> my_clients.get_rows(indexes)
[[1, u'Piotr', u'Macuk', u'Starowiejska 25/2 81-465 Gdynia',
u'piotr@macuk.pl', datetime.date(2004, 11, 30), 2,
datetime.date(2001, 1, 5), 35], [2, u'Adam', u'Nowak',
u'Grunwaldzka 117/3 80-334 Sopot', u'adam.nowak@yahoo.com',
datetime.date(2004, 11, 30), 1, datetime.date(2002, 8, 2), 25]]
```

To build mentioned system you should define the class which will use the `itools.csv` module and give you some simple API to your problem. After that you should write some scripts that use defined class.

You can find the example system in the `examples/csv_handler/my_clients.py` and `examples/csv_handler/my_system.py` files. The first file is the class definition with: get, add, delete, modify, reminders and save method. All methods use the client_id column to indicate the client. The second one is the example system script. It uses the MyClients class. The source code has a lot of comments and you shuold analyze it.

Of course the above example system is not complete. System should register payments and change the last_payment_date value and probably modify the discount rate accordingly. It is simple enhancement and you could do it by yourself if you need the system for small ISP company.

The source code used in that chapter is placed in `examples/csv_handler/example.py` file.

4.3 CSV handler API

`get_row(index)`
- Return row indexed by index.

`get_rows(indexes)`
- Return rows indexed by indexes.

`get_nrows()`
- Return number of rows in csv file.

`get_all_rows()`
- Return all csv file rows.

`add_row(row)`
- Append the new row.

`del_row(index)`
- Delete row indexed by index.

`del_rows(indexes)`
- Delete rows indexed by indexes.

`get_columns_by_names(columns)`
- Return only selected columns by its names (names in self.columns).

`get_columns_by_indexes(columns)`
- Return only selected columns by numerical indexes.

`search(query)`
- Return list of row indexes returned by executing the query or None when one or more query items is None (query item is None when the query item column is not indexed). The query item can be:

  - the (column_name, value) tuple
  - the operator: 'or' or 'and'
  - the list of the previous advanced_search result.

The query parameter is list of query items for example:

```
- [('name', 'dde')]
- [('name', 'dde'), 'and', ('country', 'Sweden')]
- [('name', 'dde'), 'or', ('name', 'fse'),
   'and', ('country', 'France')]
- [result1, 'and', result2, 'or', result3]
```

Chapter 5

Datatypes

Introduction

Information stored in a file or sent through a network appears as a chain of bytes, but it represents high level information. For example we may want to keep some basic data about a master piece book:

```
<metadata xmlns:dc="http://purl.org/dc/elements/1.1/">
  <dc:title>100 años de soledad</dc:title>
  <dc:creator>Gabriel Garcia Marquez</dc:creator>
  <dc:date>1999-07-02</dc:date>
  <dc:language>es</dc:language>
</metadata>
```

Above we have used the XML language to keep the title, author, date of publication and language of our book; but we may prefer a different file format:

```
BEGIN:RECORD
  TITLE:100 años de soledad
  CREATOR:Gabriel Garcia Marquez
  DATE:1999-07-02
  LANGUAGE:es
END:RECORD
```

Indepently of the format used, one thing is sure, we would like to load the information with a type:

```
'100 años de soledad'     -(unicode)->  u'100 años de soledad'
'Gabriel Garcia Marquez'  -(unicode)->  u'Gabriel Garcia Marquez'
'1999-07-02'              ----(date)->  datetime.date(1999, 7, 2)
'es'                      --(string)->  'es'
```

This process called *deserialization*. The opposite operation, which we are also interested in, is called *serialization*: to transform a value to a chain of bytes.

Other operations we may be interested include:

- To check wether a string value is correct; for example, to check wether the date respects the ISO 8601 standard.

- To provide a default value when a field is missing; for example we may assume the language is English if not specified otherwise.

The packages `itools.datatypes` and `itools.schema` provide an infrastructure to do all this and more. What is useful not only to serialize and deserialize files, but also for other purposes, like validating user input data.

5.1 Datatypes

In Python there are basic types like `unicode`, `integer` or `float`. And the there are more complex types like dates.

The module `itools.datatypes` provides an infrastructure orthogonal to the Python types. The basic service provided by this infrastructure is the deserialization and serialization of values; which is implemented as the couple of class methods `decode` and `encode`, for example:

```
>>> from itools.datatypes import DateTime
>>> datetime = DateTime.decode('2005-05-02 16:47')
>>> datetime
datetime.datetime(2005, 5, 2, 16, 47)
>>> DateTime.encode(datetime)
'2005-05-02 16:47'
```

This approach, to implement the serialization/deserialization code separate from the type itself, allows to avoid subclassig built-in types, what has a performance impact.

This also illustrates one of the software principles behind the itools coding, different programming aspects should be cleary distinct in the implementation and programming interface.

5.1.1 Out of the box

Out-of-the-box `itools.datatypes` provides support for the following types:

Integers (`Integer`) An integer number is serialized using ASCII characters. This means a call to `Integer.decode(x)` is equivalent to `int(x)`, and `Integer.encode(x)` does the same than `str(x)`.

Text (`Unicode`) Text strings are serialized using the UTF-8 encoding (by default).

Byte strings (`String`) A byte string does not needs to be serialized or deserialized, the output is always equal to the input.

Booleans (`Boolean`) Boolean values are encoded with the "0" character for the *false* value and with the "1" character for the *true* value.

Dates (`Date`) Dates are encoded following the ISO 8601 standard[1]: `YYYY-MM-DD`.

Datetimes (`DateTime`) Date and time is encoded with the pattern: `YYYY-MM-DD hh:mm`.

[1] http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html

URIs (`URI`) The URI decoder will build and return one of the URI reference objects defined in the `itools.uri` package, usually it will be an instance of the class `itools.uri.generic.Reference`.

Filenames (`FileName`) Usually filenames include extensions to indicate the file type, and sometimes other information like the language. The filename decoder will parse a filename and return a tuple where the first element is the filename, the second element is the file type, and the last element is the language. For example:

```
>>> from itools.datatypes import FileName
>>> FileName.decode('index.html.en')
('index', 'html', 'en')
>>> FileName.decode('index.html')
('index', 'html', None)
>>> FileName.decode('index')
('index', None, None)
```

XML qualified names (`QName`) An XML qualified name has two parts, the prefix and the local name, so our decoder will return a tuple with these two elements:

```
>>> from itools.datatypes import QName
>>> QName.decode('dc:title')
('dc', 'title')
>>> QName.decode('href')
(None, 'href')
```

The encoder expects a two element tuple:

```
>>> QName.encode(('dc', 'title'))
'dc:title'
>>> QName.encode((None, 'href'))
'href'
```

5.1.2 Defining new datatypes

There are two ways to define a new datatype: subclassing and instanciating.

Subclassing

All datatytpes inherit from the abstract class `DataType`. To define a new datatype either subclass directly from `DateType`, or from any other subclass of it.

As an example we are going to define a datatype that loads mimetypes as a two elements tuple:

```
import mimetypes
from itools.datatypes import DateType

class MimeType(DateType):
```

```
    default = None

    @staticmethod
    def decode(data):
        return tuple(data.split('/'))

    @staticmethod
    def encode(value)
        return '%s/%s' % value

    @staticmethod
    def is_valid(data):
        return mimetypes.guess_extension(data) is not None
```

Two things to highligth:

- We have set the default value to `None`, though this is not really needed since the `DataType` class already defines this variable to `None`.

  Another good default value maybe `('application', 'octet-stream')`.

- We have added the method `is_valid`, which is not defined by any other datatype included in `itools`. This illustrates that the datatypes can be extended with whatever logic, which we could use later in the application code.

5.1.3 Instanciating

This is a more compact way to specialize a datatype, when the changes are small. For example:

```
from itools.datatypes import String

WorkflowState = String(default='private')
```

Here we have defined a workflow state as an string, whose default value is `private`.

Note that a shortcoming of this approach is that, unlike subclassing, it is not possible to instanciate a datatype that already is an instance.

Chapter 6

Gettext

Chapter 7

Handlers

Here we use the term *handlers* as a shorthand for *resource handlers*; as this expression suggests the package `itools.handlers` is closely related to the package `itools.resouces` (see Chapter 11).

For remembering purposes, `itools.resources` provides an abstraction layer for files and folders, regardless of where these files and folders are stored (in the filesystem, in a Web or FTP server, etc.).

Now we can define a *resource handler* as:

> *an object consisting of (1) a data structure that represents the resource it handles, and (2) a programming interface to access and modify this data structure*

In other words: resources provide persistence to our data, while handlers provide semantics. So for example we have handler classes to manage XML or CSV files, and we can write our own handler classes not only to implement other file formats, but also to specialize the behaviour of our handlers for whatever purpose.

As happens with resources, there are two kinds of handlers, one for files, the other for folders.

7.1 File handlers

To get a feeling, let's start with an example:

1. First we load a resource:

```
>>> from itools.resources import get_resource
>>> resource = get_resource('http://example.com')
```

2. Now we build a handler for the resource:

```
>>> from itools.html import HTML
>>> handler = HTML.Document(resource)
>>> print handler
<itools.html.HTML.Document object at 0x40647b4c>
>>> print handler.to_str()
```

```
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing
...
```

There are several things to highlight here. First the way we have built our handler instance, passing a resource to a handler class (`HTML.Document`), later we will see other ways to build handler instances.

Second the `to_str` method (all the file handlers have this method), which serializes the handler, i.e. transforms the handler to a sequence of bytes. This is a key concept to which we will come back later.

3. Now we can start working with the handler:

```
>>> from itools.xml import XML
>>> for node in handler.traverse():
...     if isinstance(node, XML.Element) and node.name == 'title':
...         print node.children
Example Web Page
```

Unlike `to_str`, the method `traverse` is specific to XML documents. Every handler class provides its own API.

One resource, many handlers

The relationship between resources to handlers is *1* to *n*. While there may be several different handlers associated to the same resource (though it is not obvious how this is useful), a handler is only associated to one resource.

The resource associated to a handler is accessible through the `resource` attribute. Following the example above, the code below checks that the resource we loaded in the first place is exactly the same our handler is associated to:

```
>>> resource
<itools.resources.http.File instance at 0x404aadec>
>>> handler.resource
<itools.resources.http.File instance at 0x404aadec>
>>> resource is handler.resource
True
```

7.1.1 Handler's state

A handler is a non-persitent object, it stores in volatile memory a data structure that represents the resource's content. For example, the Figure 7.1 shows at the left an XML file, and at the right the state of the handler as a tree of XML elements.

The state of a handler is stored within the instance variable `state`, what this variable contains depends on the handler class; e.g. an element tree for XML documents, a mapping from message to translation for PO files, or just a unicode string for plain text files. For example:

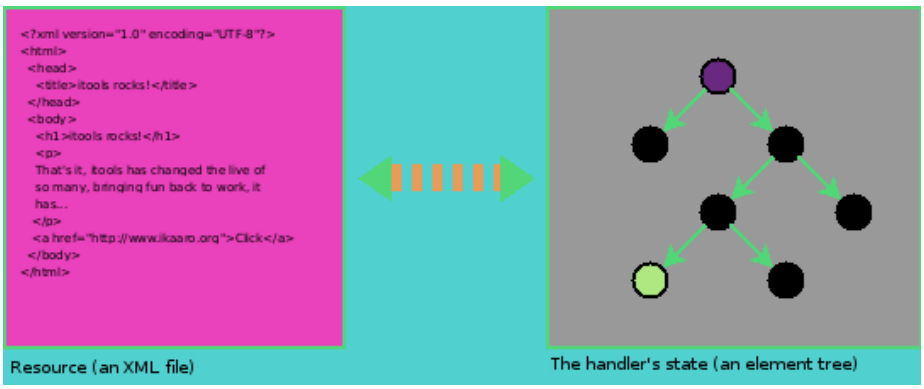

Figure 7.1: Handler's state

```
>>> from itools.xhtml import XHTML
>>> handler = XHTML.Document()
>>> print handler
<itools.xhtml.XHTML.Document object at 0xb7cae8ac>
>>> print handler.state
<itools.handlers.Handler.State object at 0xb7cae92c>
>>>
>>> from pprint import pprint
>>> pprint(handler.state.__dict__.keys())
['xml_version',
 'source_encoding',
 'document_type',
 'root_element',
 'standalone']
```

The state variable should never be accessed directly, instead the API each handler provides should be used.

We call *load* to the process of building the handler state from the data stored in the resource. And viceversa, we call *save* to the process of updating the resource with the changes made to the handler state.

So, while the handler and the resource are synced inmediately after the handler is loaded, this is to say, they contain the same information, one or the other may change, hence becoming desynchronized. There are a couple of scenarios we want to deal with:

- Once the handler is loaded, the resource is modified by other means, for example you open the file with an editor and modify it.

  Then the handler is outdated, its state represents an older version of the resource.

- After loading the handler, its state is modified through the API it provides.

  Then the resource is outdated, as the handler contains a newer version of the information.

Load

It may happen that the resource (a web page in our example) gets updated, then the handler will keep a data structure on memory that does not correspond anymore to the content of the resource, in other words, the handler would be out-of-date. We can check this by comparing the last modification time of the resource with the timestamp every handler always keeps:

```
>>> print handler.resource.get_mtime()
2004-11-28 19:53:57
>>> print handler.timestamp
2004-12-29 19:31:39.055367
>>> handler.resource.get_mtime() > handler.timestamp
False
```

Ok, the example above shows everything is alright. But what to do if it was not? then we would need to reload the resource, this is done with the `load_state` method:

```
>>> print handler.timestamp
2004-12-29 19:31:39.055367
>>> handler.load_state()
>>> print handler.timestamp
2004-12-29 19:51:50.152499
```

You should try it yourself with a resource in the filesystem: start the Python interpreter, build a handler for a resource in the filesystem, modify the resource with another program, see how the resource modification time is more recent than the handler's timestamp, then reload the handler and verify the handler is up-to-date now.

The full prototype of the method `load_state` follows:

`load_state(resource=None)` - Loads the data from the given resource into the handler, if no resource is given, the one the handler is attached to will be used.

As you see the `load_state` method accepts an optional parameter, a resource. If it is not given the handler will be reloaded from the resource it handles. But if other resource is passed it will be loaded from the given resource. Then we would reach a situation opposite to the one seen before: the resource would be outdated, as the handler would keep a newer version.

Save

Another way to get a handler more recent than the resource it is associated with, is to modify the handler through the API it offers, which depends on the handler class (e.g. the API of an XML handler is different from the API of an image handler).

To update the resource so both resource and handler are in sync again we use the `save_state` method:

`save_state()` - Saves the handler into the resource.

So `load_state` and `save_state` are two sides of the same coin. The first one reloads the handler with the resource, the second one saves the handler state in the resource.

Serialization and de-serialization

A file resource represents a sequence of bytes. A handler keeps a data structure in memory, and offers an API to inspect and modify this data structure.

Two key concepts in computer science will help to clarify what we have seen up to now:

serialization is the process by which a sequence of bytes is transformed into a data structure.

de-serialization is the process by which a data structure is transformed to a sequence of bytes.

Now it becomes obvious that the `load_state` method what actually does is to de-serialize the resource and update the handler. And viceversa, the `save_state` method serializes the handler and updates the resource.

Remember the method `to_str` we saw at the beginning? it is the one responsible for the serialization process. The `save_state` method calls `to_str` first, and then updates the resource.

7.1.2 The skeleton

As we have seen the handler constructor expects one parameter: the resource it is meant to handle (and once the handler is built the resource is always accessible with `handler.resource`).

However, it is also possible to build a handler without passing it any parameter; in this case a memory resource will be built on the fly, let's see an example:

```
>>> from itools.html import HTML
>>> handler = HTML.Document()
>>> handler
<itools.handlers.HTML.Document object at 0x403ee12c>
>>> handler.resource
<itools.resources.memory.File instance at 0x40638e4c>
>>> print handler.to_str()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
    <title></title>

  <body></body>
</head></html>
```

The default content of the resource depends on the handler, and it is called the *handler skeleton*. The constructor also accepts arbitrary keyword parameters:

```
>>> handler = HTML.Document(title='Hello World')
>>> print handler.to_str()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
    <title>Hello World</title>
  </head>
  <body></body>
</html>
```

The keyword parameters are used to initialize the skeleton; in the example above the `title` parameter defines the HTML document title, which by default is empty. The parameters accepted depend on the handler.

7.1.3 Text and binary handlers

File handlers split into text and binary handlers. The class `Text` is the base class for all the text handlers.

The key difference is that text handlers represent the resource's content in memory as a unicode string, instead of just a byte string.

When the resource is loaded the text handler tries to guess the character encoding of the resource's content. For some file formats this information is stored within the content itself; for example an XML document starts by an XML declaration that specifies the encoding, if the declaration is missing it is assumed the encoding is UTF-8. When this information is not explicitly indicated, the text handler tries to guess the encoding by brute force.

With the character encoding the text handler decodes and loads the resource's content.

The API of binary and text handlers also differs, compare the one offered by the `File` handler class:

> `to_str()`
> - Returns the resource data as a byte string (this is similar to the method `handler.resource.read()`).
>
> `set_data(data)`
> - Updates the handler state with the given data.

with the one offered by the `Text` handler class:

> `to_str(encoding='UTF-8')`
> - Returns the resource data as a byte string, using the given encoding (defaults to `UTF-8`). Note that this will be different than the string returned by `handler.resource.read()` if the resource data is not encoded in `UTF-8` in the source.
>
> `to_unicode(encoding='UTF-8')`
> - Returns the resource data as an unicode string.

Note the `to_str` method for text handlers accepts the encoding as an optional parameter. This not only will return a byte string in the given charater encoding,

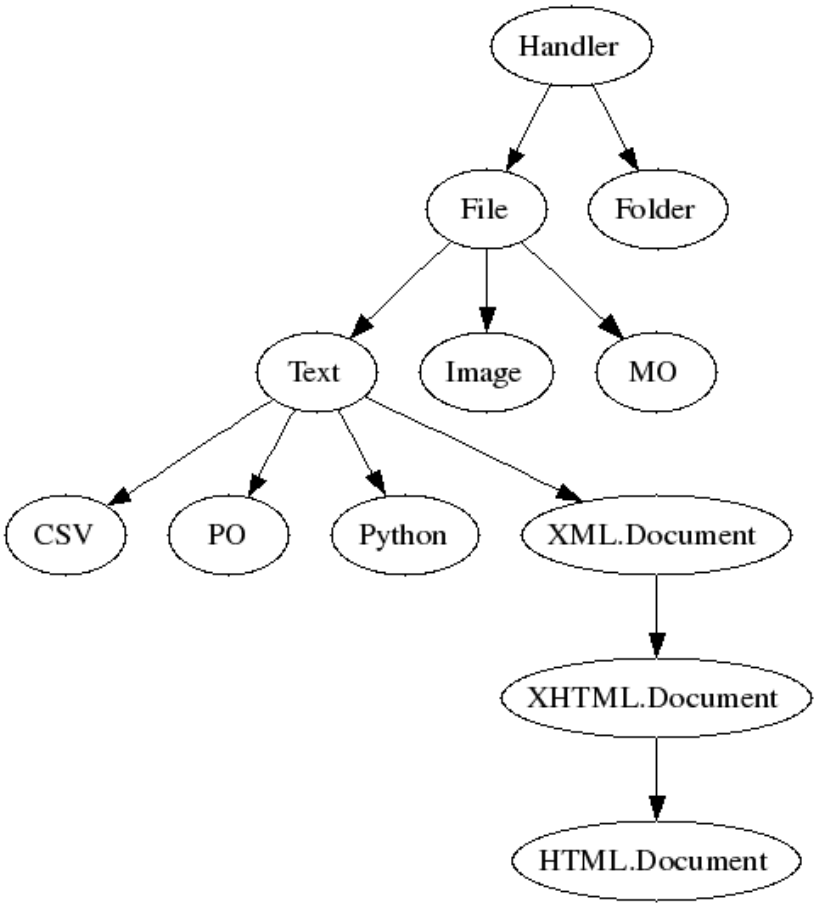

Figure 7.2: The handler tree

it will also specify the encoding in the byte string for file formats that should include it.

There is also the new method `to_unicode`, which returns the content as a unicode string. This method accepts the encoding parameter too. For file formats that do not include the encoding within the content, the result of `to_unicode` will be the same regardless of the given encoding. But for file formats like XML that should specify the encoding within the content, the returned unicode string will include it.

7.1.4 Overview of the available handlers

Out of the box `itools` comes with several handlers for different standard file formats. The Figure 7.2 shows an excerpt of the tree of the available handler classes, which express the inheritance relationship between them.

We are not going to see folder handlers here, as they will be studied in the next chapter.

We have already seen the difference between binary and text handlers. Now

we are going to present some of the higher level file handler classes `itools` includes.

XML.Document This is the default handler for XML documents, which internally are represented as a tree. The API includes the methods already described for the `Text` handler; specific methods of `XML.Document` are:

`__cmp__(other)`
- Lets to compare two XML documents.

`get_root_element()`
- Returns the root element of this XML document.

`traverse()`
- This method allows to traverse the XML document, as it has a tree structure. It is a generator which returns a node at a time, starting by the document instance.

`traverse2()`
- As `traverse`, this method allows to traverse the XML document, though it is more powerful, and complex. It will be explained in the XML chapter.

XHTML.Document The handler for XHTML documents, it extends the API provided by `XML.Document` API with the methods:

`get_head()`
- Returns the head element.

`get_body()`
- Returns the body element.

`to_text()`
- Strips all the XML markup and returns the text content of the XHTML document. This is useful, for example, to index the document.

PO The handler to manage PO files, the message catalog of the GNU gettext utilities.

`get_msgids()`
- Returns the list of message ids stored in the catalog.

`get_messages()`
- Returns the list of messages stored in the catalog, where each message is represented as an instance of the class `PO.Message`.

`get_msgstr(msgid)`
- Returns the message string for the given message id.

`set_message(msgid, msgstr=[u''], comments=[u''], references={})`
- Adds a message (from the given parameters) to the catalog.

7.1.5 The handler factory

So far we have built a handler instance through a handler class:

```
>>> handler = HTML.Document(resource)
>>> handler
<itools.xml.HTML.Document object at 0x405740cc>
```

This is the standard pattern to build instances. However, if the resource is not an HTML document this procedure would fail. In other words, this procedure is only useful if you already know the kind of resource you are working with.

Another option is to let `itools` to choose which handler class to use. This can be done with the `build_handler` method, which provides the factory pattern.

`build_handler(resource)` - A class method that identifies the given resource, chooses the available handler class that better matches it, and builds and returns a handler instance for it.

For example, from the `examples` directory type:

```
>>> from itools.resources import get_resource
>>> from itools.handlers.Handler import Handler
>>> from itools import xml
>>>
>>> here = get_resource('.')
>>> for name in here.get_resource_names():
...     resource = here.get_resource(name)
...     handler = Handler.build_handler(resource)
...     print name, handler
...
chapter6 <itools.handlers.Folder.Folder object at 0x40538d4c>
chapter7 <itools.handlers.Folder.Folder object at 0x4032c0cc>
hello.txt <itools.handlers.Text.Text object at 0x40536f4c>
hello.xhtml <itools.xml.XHTML.Document object at 0x4053b70c>
```

Note that `build_handler` is a class method. In the example above we have called it through the most abstract handler class: `Handler`, which is the root of the inheritance tree. But it is also possible to call it with another handler class:

```
>>> from itools.xml import XML
>>>
>>> resource = here.get_resource('hello.xhtml')
>>> XML.Document.build_handler(resource)
<itools.xml.XHTML.Document object at 0x405c6ecc>
```

There is an important difference between calling `build_handler` from one or another class: the set of possible handler classes to use is restricted to all the sub-classes of the choosen handler class. For example, if we pass a plain text file to `XML.Document.build_handler`, it will fail:

```
>>> resource = here.get_resource('hello.txt')
>>> XML.Document.build_handler(resource)
Traceback (most recent call last):
  [...]
xml.parsers.expat.ExpatError: syntax error: line 1, column 0
```

The `get_handler` shorthand

There is a short way to load a handler (instead of loading first the resource and then building the handler explicitly), the function `get_handler`:

`get_handler(uri)` - Loads the resource at the given uri, tries to guess its mimetype by different meanings (name extension, etc.), searches for a suitable handler class in the registry, builds and returns the handler for the resource.

Compare the explicit way seen before:

```
>>> from itools.resources import get_resource
>>> from itools.handlers.Handler import Handler
>>>
>>> resource = get_resource('http://example.com')
>>> handler = Handler.build_handler(resource)
```

With the shorthand:

```
>>> from itools.handlers import get_handler
>>>
>>> handler = get_handler('http://example.com')
```

7.2 Writing file handler classes

The chapter before we have learnt about file handlers and how to use them, now we are going to learn how to write our own handler classes, what by the way will help to solidify the concepts seen before.

The explanation will be driven by an example: we are going to write a task tracker. The code can be found in the directory `examples/chapter6`.

7.2.1 Functional scope

Lets start by defining the functional scope of our task tracker. It is going to be very simple, it will be a collection of tasks where every task will have three fields:

- *title*, a short sentence describing the task.
- *description*, a longer description detailing the task.
- *state*, it may be *open* (if the task has not been finished yet), or *closed* (if the task has been finished).

The task tracker will provide an API to manipulate the collection of tasks: create a new task, see either the open or the closed tasks, and close a task.

7.2.2 The file format

Now that we know what we want to do, we have to decide where and how the information will be stored.

We will keep the tasks in a single text file, with a format somewhat similar to the one used by the standards *vCard* and *iCal*, for example:

```
title:Re-write the chapter about writing handler classes.
description:A new chapter that explains how to write file handler
 classes must be written, it should go inmediately after the chapter
 that introduces file handlers.
state:closed

title:Finish the chapter about folder handlers.
description:The chapter about folder handlers needs much more work.
 For example the skeleton of folder handlers must be explained.
state:open
```

Each task is separated from the next one by a blank line. Every field starts by the field name followed by the field value, both separated by a colon. If a field value is very long it can be written in multiple lines, where the second and next lines start by an space.

This very same file can be found in the examples directory with the name `itools.tt`. Using our own filename extension (`tt`) will prove useful, as we will see later.

7.2.3 De-serialization

The first draft of our handler class will be able to load (de-serialize) the resource into a data structure on memory.

```
from itools.handlers.Text import Text


class Task(object):
    def __init__(self, title, description, state='open'):
        self.title = title
        self.description = description
        self.state = state


class TaskTracker(Text):

    def _load_state(self, resource=None):
        # Load the resource as a unicode string
        Text._load_state(self, resource)
        # Split the raw data in lines.
        state = self.state
        lines = state.data.splitlines()
        # Append None to signal the end of the data.
        lines.append(None)
        # Free the un-needed data structure, 'state.data'
        del state.data
```

```
        # Initialize the internal data structure
        state.tasks = []
        # Parse and load the tasks
        fields = {}
        for line in lines:
            if line is None or line.strip() == '':
                if fields:
                    task = Task(fields['title'],
                                fields['description'],
                                fields['state'])
                    state.tasks.append(task)
                    fields = {}
            else:
                if line.startswith(' '):
                    fields[field_name] += line
                else:
                    field_name, field_value = line.split(':', 1)
                    fields[field_name] = field_value
```

First, our handler class `TaskTracker` inherits from the handler class `Text`, because it is intended to manage a text file.

The method `_load_state` is the one to implement, it is responsible to deserialize the resource and build a data structure on memory that represents it.

The first thing it does is to call the parent's `_load_state` method, which will read the resource's data and get a unicode string from it, which will be stored in the `self.state.data` variable. This is an intermediary representation, which will be discarded.

The rest of the method process the raw data and builds a list of instances of the class `Task` into the variable `self.state.tasks`.

Lets try it:

```
>>> from pprint import pprint
>>> import textwrap
>>> from itools.resources import get_resource
>>> from TaskTracker import TaskTracker
>>>
>>> resource = get_resource('itools.tt')
>>> task_tracker = TaskTracker(resource)
>>>
>>> pprint(task_tracker.state.tasks)
[<TaskTracker.Task object at 0xb7aebd4c>,
 <TaskTracker.Task object at 0xb7aebe6c>]
>>>
>>> task = task_tracker.state.tasks[0]
>>> print task.title
Re-write the chapter about writing handler classes.
>>> print textwrap.fill(task.description)
A new chapter that explains how to write file handler classes must be
written, it should go inmediately after the chapter that introduces
file handlers.
>>> print task.state
closed
```

7.2.4 Serialization

Now we are going to write the other half, the serialization process, just adding the `to_unicode` method to the `TaskTracker` class:

```
def to_unicode(self, encoding='UTF-8'):
    lines = []
    for task in self.state.tasks:
        lines.append(u'title:%s' % task.title)
        description = u'description:%s' % task.description
        description = textwrap.wrap(description)
        lines.append(description[0])
        for line in description[1:]:
            lines.append(u' %s' % line)
        lines.append(u'state:%s' % task.state)
        lines.append('')
    return u'\n'.join(lines)
```

Note that we implement the `to_unicode` method instead of `to_str` because this is a text handler. The `encoding` parameter is not used in our example because the file format does not specifiy the encoding within its content, but it must be declared anyway.

Lets try our new code:

```
>>> print task_tracker.to_str()
title:Re-write the chapter about writing handler classes.
description:A new chapter that explains how to write file handler
 classes must be written, it should go inmediately after the chapter
 that introduces file handlers.
state:closed

title:Finish the chapter about folder handlers.
description:The chapter about folder handlers needs much more work.
 For example the skeleton of folder handlers must be explained.
state:open
```

We could have tried `to_unicode` to get a similar output, except that then the result would be a unicode string instead of a byte string.

7.2.5 The API

Now it is time to write the API to manage the tasks, here is an excerpt:

```
def add_task(self, title, description):
    task = Task(title, description)
    self.state.tasks.append(task)


def show_open_tasks(self):
    for id, task in enumerate(self.state.tasks):
        if task.state == 'open':
            print 'Task #%d: %s' % (id, task.title)
            print
            print textwrap.fill(task.description)
```

```
            print
            print


    def close_task(self, id):
        task = self.state.tasks[id]
        task.state = u'closed'
```

The first method, `add_task` creates a new task, whose state will be *open*. The method `show_open_tasks` prints the list of open tasks with a human readable format (we could write a method that returns HTML instead, to use our task tracker on the web). Finally, the method `close_task` closes the task.

7.2.6 The skeleton

The skeleton for our task tracker should be empty, but to illustrate this feature we are going to implement an skeleton with one dummy task:

```
    def get_skeleton(self):
        return 'title:Read the docs!\n' \
               'description:Read the itools documentation, it is\n' \
               ' so gooood.\n' \
               'state:open\n'
```

To exercise the whole thing we are going to create a new task tracker, we will close the first task, add a new one, and look what we have.

```
>>> from TaskTracker import TaskTracker
>>>
>>> task_tracker = TaskTracker()
>>> task_tracker.show_open_tasks()
Task #0: Read the docs!

Read the itools documentation, it is so gooood.


>>> task_tracker.close_task(0)
>>> task_tracker.add_task('Join itools!',
...   'Subscribe to the itools mailing list.')
>>> task_tracker.show_open_tasks()
Task #1: Join itools!

Subscribe to the itools mailing list.
```

Now, don't forget to save the task tracker in the file system, so you can come back to it later:

```
>>> from itools.handlers import get_handler
>>>
>>> tmp = get_handler('/tmp')
>>> tmp.set_handler('most_important_things_in_live.tt', task_tracker)
>>> tmp.save_state()
```

7.2.7 Register

However:

```
>>> task_tracker = tmp.get_handler('most_important_things_in_live.tt')
>>> print task_tracker
<itools.handlers.File.File object at 0xb7c00f0c>
```

It would be nice if the code above worked. To achieve it we will associate the new mimetype `text/x-task-tracker` to the file extension `tt`, we will tell our handler class is able to manage that mimetype with the variable class `class_mimetypes`, and we will register our handler class to its parent:

```
import mimetypes
mimetypes.add_type('text/x-task-tracker', '.tt')


class TaskTracker(Text):

    class_mimetypes = ['text/x-task-tracker']

    [...]


Text.register_handler_class(TaskTracker)
```

And *voilà*:

```
>>> task_tracker = tmp.get_handler('most_important_things_in_live.tt')
>>> print task_tracker
<TaskTracker.TaskTracker object at 0xb7af084c>
```

The full code can be found in `examples/chapter6/TaskTracker.py`.

7.3 Folder handlers

A handler that deserves its particular chapter is the default handler for folders.

7.3.1 Folder's state

If the content of a file resource is a byte string, the content of a folder resource is a set of resources, each one identified by a name. Hence, the state of a folder handler is a mapping fron handler name to handler instance.

But imagine a folder that contains many big files. To load its state would mean to load all the files it contains, what is unacceptable from a performance point of view. This is the reason the folder handler implements lazy load, this is to say, it loads a handler only when it is needed.

The folder state is stored in the variable `cache`, because it behaves like a cache. See:

```
>>> examples = get_handler('examples')

[some operations later]
```

```
>>> pprint(examples.state.cache)
{'.arch-ids': None,
 'chapter8': <itools.handlers.Folder.Folder object at 0xb77b2fac>,
 'chapter9': None,
 'hello.txt': <itools.handlers.Text.Text object at 0xb77b2bcc>,
 'hello.xhtml': None,
 'task_tracker0': None,
 'task_tracker1': None,
 'task_tracker2': None}
```

As the code above shows, a handler that has not yet been loaded appears in the cache with the `None` value.

7.3.2 The API

The programming interface for folder handlers remembers the one of folder resources, where there we spelled `get_resource` here we will write `get_handler`. Details follow:

`get_handler(path)`
- Returns a handler for the resource at the given path. It will use the available handler class that better matches the resource mimetype.

`get_handler_names(path='.')`
- Returns a list with the names of all the handlers in the given path.

`get_handlers(path='.')`
- Returns the handlers in the given path (it is a generator).

`has_handler(path)`
- Returns `True` if there is a handler in the given path, `False` otherwise.

`set_handler(path, handler)`
- Adds the given handler to the given path. Actually what is added is the resource associated to the handler.

`del_handler(path)`
- Removes the handler at the given path (i.e. the associated resource).

7.3.3 Example

As the proverb says, *a code snippet is worth more than one thousand words*:

1. First we build a handler for the temporary directory:

```
>>> from itools.handlers import get_handler
>>>
>>> tmp = get_handler('/tmp')
>>> tmp
<itools.handlers.Folder.Folder object at 0x40652dec>
```

```
>>> tmp.resource
<itools.resources.file.Folder instance at 0x406acacc>
```

2. Second, we create a new HTML handler:

```
>>> from itools.xml import HTML
>>> hello = HTML.Document(title='Hello World')
>>> hello.resource
<itools.resources.memory.File instance at 0x405e49cc>
```

   Note that the associated resource is built on the fly and lives in memory.

3. Third, we set the HTML handler to the temporary folder:

```
>>> tmp.set_handler('hello.html', hello)
```

   What this actually does is to add the file `hello.resource` (which lives in memory) to the folder `tmp.resource` (which is on the file system); this is to say, it creates a new file in the file system at `/tmp/hello.html`.

4. Finally, we get the handler we just added:

```
>>> hello = tmp.get_handler('hello.html')
>>> hello.resource
<itools.resources.file.File instance at 0x40638c6c>
```

   Note that the handler `hello` we have built in these last lines manages a resource that lives in the file system.

7.3.4 The handler tree

Folders allow to classify files, hence giving a tree structure to our data. Every handler has two attributes, `parent` and `name`, they tell us where the handler is in the handler tree:

```
>>> hello.parent
<itools.handlers.Folder.Folder object at 0x403ebb2c>
>>> hello.name
'hello.html'
>>> hello.parent is tmp
True
>>> print tmp.parent
None
>>> print tmp.name

>>>
```

Based on these two attributes handlers provide the following API:

`get_abspath()`
- Returns the absolute path from the tree root to the `self` handler.

`get_root()`
- Returns the handler for the root of the tree.

`get_pathtoroot()`
- Returns a relative path from `self` to the tree root (e.g. `../../..`).

`get_pathto(handler)`
- Returns a relative path from `self` to the given handler (which is supposed to be in the same tree), for example: `../../zoo/lion`.

`traverse()`
- This method allows to traverse the handler tree below this folder. It is a generator which returns a handler at a time, starting by this folder.

`acquire(name)`
- If the current handler is a folder and contains a resource with the given name, then return a handler for it; otherwise look at the parent folder, and recursively to the root tree. This method actually shows how to implement *acquisition*.

Chapter 8

HTML

Chapter 9

Internationalization (i18n)

The task tracker from the previous chapter provides a user interface in only one language. The purpose of this chapter is to explain how to build applications that provide a user interface in many languages. We will illustrate this building up on the task tracker example.

See the file layout of the already multilingual task tracker:

```
Makefile
TaskTracker.py
TaskTracker_view.xml.en
locale/
  locale.pot
  en.po
  es.po
```

There are two things to note. First the `locale` directory: it is a database which keeps the translations for the text messages of the user interface. The translations are stored in *PO*[1] files, one per language; in our example there is one for English (`en.po`) and another for Spanish (`es.po`). Each *PO* file keeps the source messages, usually written in English, and their translations (because the source language is English, the `en.po` file will be usually empty).

The second thing to note is the `Makefile`, it will help us to automatize the localization and build processes. The Figure 9.1 shows these two processes.

9.1 Internationalization

Internationalization:

> *the operation by which a program, or a set of programs turned into a package, is made aware of and able to support multiple languages.*

This is to say, if you have a monolingual product like the task tracker from the previous chapter, you will need to make some changes to the code, so it becomes able to deal with multiple languages. This process is called *internationalization*. Of course, you may also write international software since the beginning.

The text messages to translate are stored in the XML templates and the Python code, so these are the two things to internationalize.

[1] XXX

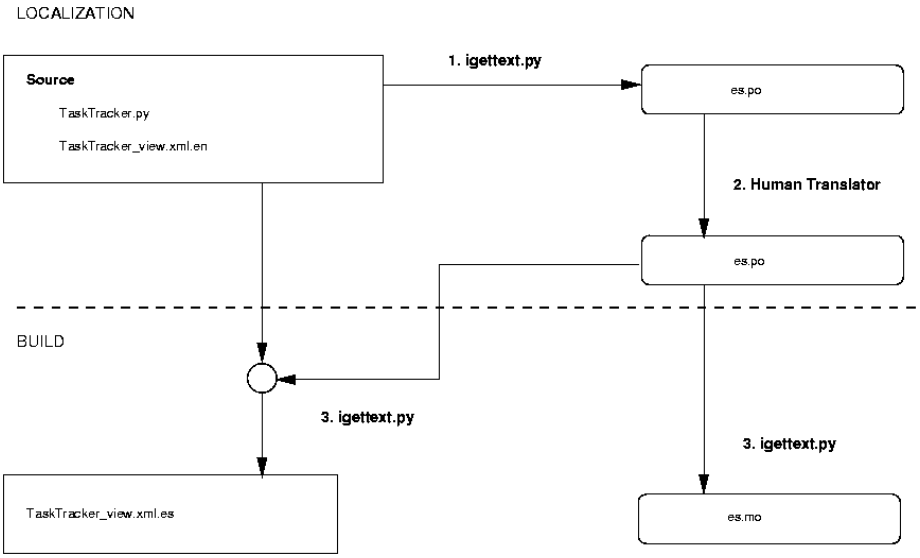

Figure 9.1: The localization process

9.1.1 Python code

To start, our task tracker must be *domain aware*:

```
from itools.gettext.domains import DomainAware

class TaskTracker(Text, DomainAware):

    class_mimetypes = ['text/x-task-tracker']
    class_domain = 'task tracker'
```

We use here the word *domain* because it is standard in the internationalization jargon. Here by a *domain* we understand a database that keeps message translations for one or more languages. A *domain aware* class is one that is implicitly associated to a domain and has an API to access that domain.

Every application and library will have a unique domain name, this will allow us to explicitly refer to a given domain to retrieve translations from it. The domain name is specified with the class variable `class_domain`.

By the way, we need to register the domain:

```
from itools import get_abspath
from itools.gettext.domains import register_domain

domain_path = get_abspath(globals(), '../locale')
register_domain(TaskTracker.class_domain, domain_path)
```

The class `DomainAware` provides the application programming interface we will use within the Python code:

`get_languages()`
- Returns a list with the language codes of our application. By default it is not implemented, so it must be overriden by subclasses.

`select_language(languages=None)`
- Chooses and returns a language from the given list of languages. If the languages are not given, the method `get_languages` will be called to get them. The default implementation uses the environment variable `LANGUAGE`.

`gettext(message, language=None, domain=None)`
- Returns the translation for the given message in the given language, from the given domain. If there is not a translation the input message is returned. If no language is given the method `select_language` will be called to choose one from the languages available in the domain. By default the domain used is the one specified in the class variable `class_domain`.

Now we just need to replace every text string in the source code by a call to `gettext`, e.g `u'hello'` becomes `self.gettext(u'hello')`. In our example it would be:

```
def show_open_tasks(self):
    for id, task in enumerate(self.state.tasks):
        if task.state == 'open':
            print self.gettext(u'Task #%(id)d: %(title)s') \
                  % {'id': id, 'title': task.title}
    ...
```

Note that we have used named arguments for the formatted string instead of positional arguments. This is because the position of the arguments may be different in another language.

9.1.2 Templates

The technique we use with `itools` requires to have one template per language, being English the master language. Each template must have a distinct name, we will achieve it by appending the language code to the end of the file:

```
TaskTracker_view.xml.en
TaskTracker_view.xml.es
```

Note that only the template in English belongs to the source, the others will be automatically built with the help of `itools`.

Now we are going to make some changes to the `TaskTracker` class. First we must override the method `get_languages`:

```
def get_languages(self):
    return ['en', 'es']
```

These are the languages of our task tracker, English and Spanish. Now we must modify the `view` method to choose the right template:

```
def view(self):
    # Load the STL template
    language = self.select_language()
    handler = get_handler('TaskTracker_view.xml.%s' % language)
    ...
```

9.2 Localization

Localization:

> *the operation by which, in a set of programs already internationalized, one gives the program all needed information so that it can adapt itself to handle its input and output in a fashion which is correct for some native language and cultural habits.*

The source is ready, now starts the localization process, it consists of two steps:

1. Message extraction.
2. Human translation.

9.2.1 Message extraction

This step consists on the parsing of the source code, the extraction of the translatable messages they contain, and the building of a *PO* file which will be the input for the human translator.

This is done with the `GNU gettext`[2] tools, which are the standard in the Free Software world for software internationalization and localization; and the script `igettext.py` provided by `itools`.

But the process is automatized by the `Makefile` so the only thing we need to do is to type:

```
$ make po
```

This will parse the Python and XHTML source and update the PO files within the `locale` directory. See below an excerpt:

```
#: TaskTracker.py:117
#, python-format
msgid "Task #%(id)d: %(title)s"
msgstr ""

#: TaskTracker_view.xml.en:0
msgid "Task Tracker"
msgstr ""
```

[2] http://www.gnu.org/software/gettext/

Marking messages

The message extraction script will pick all text strings[3] from the Python source, and all text nodes from the source templates. Then it will split the messages into sentences, a process known as segmentation, it will be these sentences which will feed the message catalog.

This means that no explicit markup is required to signal a text string as translatable. This way we reduce the burden on the developer, who just needs to remember to use Python unicode strings for text, i.e. to type:

```
u'Hello world.'
```

instead of:

```
'Hello world.'
```

For the templates, the only rule is to write them properly. In particular, to use block elements as block elements, and inline elements as inline elements.

9.2.2 Human translation

The human translator receives the PO file, and must return the same PO file, but including the translations for every message. In our example the output should be:

```
#: TaskTracker.py:117
#, python-format
msgid "Task #%(id)d: %(title)s"
msgstr "Tarea #%(id)d: %(title)s"

#: TaskTracker_view.xml.en:0
msgid "Task Tracker"
msgstr "Gestor de tareas"
```

To do this work the translator don't needs to understand the file format, there are graphical tools available to do the job like:

- KBabel, `http://www.kde.org`
- poedit, `http://poedit.sourceforge.net`

9.3 Build

To close the process, we must generate the translated templates (`TaskTracker_view.xml.es`) and the *MO* files (`en.mo`, `es.mo`) that will be used in runtime by the `gettext` method.

To build the templates from the master template (`TaskTracker_view.xml.en`) and from the translated PO file (`es.po`), is done with the script `igettext.py`:

```
$ igettext.py --xhtml addressbook_view.xml.en es.po \
      > addressbook_view.xml.es
```

[3]Text strings are most often known as unicode strings in Python.

The MO file is a binary version of the PO file, and is built with the `msgfmt` tool.

But, again, the process is automatized by the `Makefile`, so the only thing we need to do is to type:

```
$ make bin
igettext.py --output=TaskTracker_view.xml.es \
  --xhtml TaskTracker_view.xml.en locale/es.po
msgfmt locale/en.po -o locale/en.mo
msgfmt locale/es.po -o locale/es.mo
touch bin
```

And *voilà*.

Chapter 10

iCalendar

Chapter 11

Resources

A resource is much like a Python file, except for two key differences:

- It may be stored anywhere, not just in the filesystem. For example, it may be in the Web, or in some FTP server, or somewhere else.
- It may not be a file, it may be a folder.

Basically, `itools.resources` provides:

> *a universal API to manage resources wherever they are stored; and by resources we mean files (a sequence of bytes) and folders (a container for other resources).*

11.1 Quick Start

There are resources that live in some well known places, they are accessed with `get_resource`:

```
>>> from itools.resources import get_resource
>>>
>>> get_resource('examples/hello.txt')
<itools.resources.file.File instance at 0x402175ac>
>>> get_resource('http://example.com')
<itools.resources.http.File instance at 0x404aadec>
```

There are resources that live in hidden places. We can build them, but we cannot reach them from the outside:

```
>>> from itools.resources import memory
>>>
>>> memory.File('hello')
<itools.resources.memory.File object at 0xb7bb528c>
```

There are files and there are folders:

```
>>> get_resource('examples/hello.txt')
<itools.resources.file.File instance at 0x402175ac>
>>> get_resource('examples')
<itools.resources.file.Folder instance at 0x401e568c>
```

11.2 File Resources (and Python Files)

A file resource is much like a Python file. Probably the most important difference is that a resource is closed by default, so it must be opened before working with it:

```
>>> resource = get_resource('http://example.com')
>>> resource.is_open()
False
>>> resource.open()
>>> resource.is_open()
True
```

And unlike a Python file a resource object can be opened and closed, and opened again and closed again, etc.

Many packages from the Standard Library and from other sources work with Python files. Usually they will be able to work with file resources too, since the API is mostly the same (just remember to open the resource first).

Here there is an slightly larger example:

```
>>> resource = get_resource('http://example.com')
>>> print resource.get_mimetype()
text/html
>>> resource.open()
>>> print resource.get_size()
438
>>> print resource.read()
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing
...
```

11.2.1 Direct access

Just for comfort file resources extend the API of Python files with methods for direct access to an arbitrary position. This is done by implementing the `sequence` interface: `__getitem__`, `__setitem__`, `__getslice__` and `__setslice__`.

This is mostly useful for binary files. Below the mandatory example:

```
>>> import struct
>>> n = struct.pack('>I', 47)
>>> resource[20:24] = n
```

This brief example shows how to write the number 47 at the position 20 of a given resource, encoded as a 32 bits unsigned integer.

For a real example you can look at `itools.catalog`, which makes extensive and systematic use of this approach.

11.3 Folder Resources

A folder is a container of other resources: files and folders. To fulfill its purpose provides a collection of methods: `get_resource`, `has_resource`, `set_resource`, etc.

Examples go:

```
>>> examples = get_resource('examples')
>>> print examples.get_resource_names()
['hello.xhtml~', 'hello.xhtml', 'hello.txt', '.arch-ids']
>>> hello = examples.get_resource('hello.txt')
>>> hello.open()
>>> print hello.read()
hello world
```

Copy a web page to the local file system

```
>>> tmp = get_resource('/tmp')
>>> web_page = get_resource('http://example.com')
>>> tmp.set_resource('example.html', web_page)
>>> copy_of_web_page = tmp.get_resource('example.html')
>>> print web_page
<itools.resources.http.File instance at 0x405e8a2c>
>>> print copy_of_web_page
<itools.resources.file.File instance at 0x405fb64c>
```

Copy a whole tree

```
>>> from pprint import pprint
>>> talks = get_resource('/home/jdavid/talks')
>>> tmp.set_resource('talks', talks)
>>> pprint(tmp.get_resource_names('talks/EuroPython2004'))
['itools-vfs',
 'Makefile',
 'itools-vfs.log',
 'itools-vfs.tex',
 'itools-vfs.aux',
 'itools-vfs.toc',
 'itools-vfs.dvi',
 'itools-vfs.ps',
 'itools-vfs.pdf',
 'itools-vfs.tex~']
```

This is more impressive when copying a big tree from one storage to another. For example we use it in the context of the **iKaaro** Content Management System to export and import web sites from the **Z**ope **O**bject **D**ata**B**ase to the file system, and back from the file system to the **ZODB**.

11.4 Resource Types

The Section 11.5 explains the general API for both files and folders. It is the purpose of this section to exaplain the types currently supported (`file`, `http`, etc.) and the differences regarding the general API.

XXX Finish

11.4.1 Memory resources

Resources in memory are very handy. They are created passing a byte string to the constructor:

```
>>> from itools.resources import memory
>>>
>>> resource = memory.File('hello world')
>>> resource
<itools.resources.memory.File instance at 0x405e458c>
>>> resource.read()
'hello world'
>>> tmp.set_resource('hello.txt', resource)
```

11.5 API

11.5.1 Package interface

- `get_resource(reference)`

  Creates a resource object for the given URI reference. The type of the resource depends on the `reference` (`itools.resources.file`, `itools.resources.http`, etc.).

11.5.2 API common to Files and Folders

Information

- `uri`

  The URI of the resource.

- `get_name()`

  Returns the name of the resource.

- `get_mimetype()`

  Returns the mimetype of the resource.

  Folder's mimetype is `application/x-not-regular-file`, always.

  For files the method used to find out the mimetype depends on the resource type (`http`, `file`, etc.). Returns `None` if it is unable to figure out the mimetype.

- `get_ctime()`

  Returns a `datetime` object with the time the resource was created. Or `None` if the creation time is unknown.

- `get_mtime()`

  Returns a `datetime` object with the last time the resource was modified. Or `None` if the modification time is unknown.

- `get_atime()`

  Returns a `datetime` object with the last time the resource was accessed. Or `None` if the access time is unknown.

Open/Close

- `open()`

  Opens the resource.

- `close()`

  Closes the resource.

- `is_open()`

  Returns `True` if the resource is open, `False` if it is closed.

Locking

- `lock()`

  Locks the resource.

- `unlock()`

  Unlocks the resource.

- `is_locked()`

  Returns `True` if the resource is locked, `False` if it is unlocked.

11.5.3 API specific to Files

- `get_size()`

  Returns the length (the number of bytes) of the resource data. Or `None` if this information is not available.

- `seek(offset, whence=0)`

- `read(size=None)`

  Returns the resource data as a byte string.

- `readline()`

- `readlines()`

- `write(data)`

  Replaces the resource content by the given data (a byte string).

- `truncate(size=None)`

- `__getitem__(index)`

  Returns the byte at the given position.

- `__getslice__(a, b)`

  Returns the byte string that goes from a to b.

- `__setitem__(index, value)`

  Sets the given value to the given position (*index*). Usually value will be just a byte, but it may be a slice too.

- `append(data)`

  Appends the given data to the resource.

11.5.4 API specific to Folders

- `get_resource(path)`

  Returns the resource in the given path (where path is either an instance of `uri.Path` or a string).

- `get_resource_names(path='.')`

  Returns a list with the names of all the resources in the given path.

- `get_resources(path='.')`

  Returns the resources in the given path (it is a generator).

- `has_resource(path)`

  Returns `True` if there is a resource in the given path, `False` otherwise.

- `set_resource(path, resource)`

  Adds the given resource to the given path.

- `del_resource(path)`

  Removes the resource at the given path.

- `del_resources(paths)`

  Removes the resources at the given paths (where paths is a list of paths).

- `traverse()`

  This method allows to traverse the resource tree below this folder. It is a generator which returns a resource at a time, starting by this folder.

- `traverse2()`

Chapter 12

RSS

Chapter 13

Schemas

Chapter 14

Simple Template Language (STL)

There are many template languages available. The *Simple Template Language* distinct attributes are:

- optimized to work with XML files[1];
- very easy to learn;
- high productivy;
- fast.

I like to define **STL** as a *descriptive* language. It is amazing how many template languages are out there that allow to put Python code inside a template, and call it a feature.

14.1 How it works

The Figure 14.1 shows how **STL** works. The input for **STL** is a source template and a Python namespace, the output is a new XML template.

There are solutions like htmlgen[2] or XX that allow to produce the XML or HTML output enterily from Python. While with tools like ZPT[3] it is possible to produce a complex web page only from a source template.

We think it is easier to read code when it is written in Python, and it is easier to read XML when it is written in XML. This is the reason **STL** not just allows, but enforces, the separation between logic and presentation.

The language **STL** don't includes any programming constructions, the four statements it provides only describe the transformations to be performed on the source template.

Lets see a simple example.

[1] **STL** is implemented as an XML namespace.
[2] XXX
[3] XXX

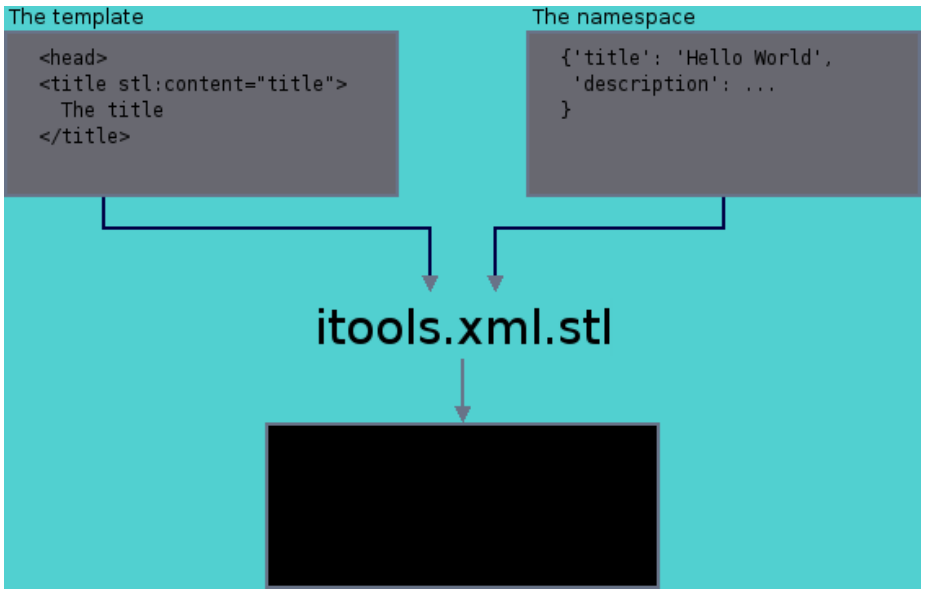

Figure 14.1: The Simple Template Language (**STL**)

14.1.1 The template

For example, look at the template below (`examples/chapter9/template.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:stl="http://xml.itools.org/namespaces/stl">
  <head></head>
  <body>
    <h1 stl:content="title" />
  </body>
</html>
```

Note the declaration of the *stl* namespace, it is mandatory since **STL** is implemented as an XML namespace.

When this template will be processed, the content of the `<h1>` tag will be replaced by the value of the variable `title`. But, where will we find `title`? Solution: in the namespace.

14.1.2 The namespace

The namespace is a mapping that is built from the Python side. Then with the template and the namespace we will just call the `stl` function to get the output. See:

```
>>> from itools.handlers import get_handler
>>> from itools.xml.stl import stl
>>>
>>> namespace = {'title': 'hello world'}     # 1. Build the namespace
>>> template = get_handler('template.xml')   # 2. Load the template
```

```
>>> print stl(template, namespace)           # 3. Go!
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns:stl="http://xml.itools.org/namespaces/stl"
      xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>hello world</h1>
  </body>
</html>
```

As the output shows, the value of the variable `title` is looked within the namespace.

14.2 The Language

14.2.1 STL attributes

There are four **STL** attributes:

- `stl:content`="*expression*"

  Replaces the element's content by the result of evaluating the given expression.

- `stl:attributes`="*name expression*[; *name expression*]*"

  For every pair "*name: expression*", replace the the value of the attribute *name* by the result of evaluating *expression*.

- `stl:if`="[not]*expression*"

  If the given expression evaluates to `True`, do nothing; if evaluates to `False`, remove the XML element.

- `stl:repeat`="*name expression*"

  The given expression is expected to be a sequence or an iterator. For every item in *expression*, create an element and add the item to the namespace stack before processing the element.

14.2.2 Expressions

The **STL** expressions are pretty simple, their syntax is:

name[/name]*

That is, a sequence of names separated by slashes. The semantics is:

1. Look the first name in the namespace stack.
2. If there are more names left, the last value found must be a namespace, then look the next name in that namespace.

   Iterate until the last name is consumed.
3. Once the end of the sequence is reached, we will have a value. If the value is callable, then call it to get a new value.

4. Finally, we should have a value that is either a string, a boolean or a sequence, depending on which statement (`content`, `repeat`, etc.) the expression is being used with.

14.3 Example: Task Tracker

Now we are going to illustrate **STL** with a more complex example. Building up on the Task Tracker from the Chapter 7.2, we are going to write a method that produces an HTML page showing all the tasks.

First, the template (see `examples/chapter9/TaskTracker_view.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:stl="http://xml.itools.org/namespaces/stl">
  <head></head>
  <body>
    <h2>Task Tracker</h2>
    <div stl:repeat="task tasks">
      <h4>
        #<stl:block content="task/id" />:
        <stl:block content="task/title" />
        (<em stl:content="task/state" />)
      </h4>
      <p stl:content="task/description" />
    </div>
  </body>
</html>
```

The first new thing this example shows is the `repeat` statement. While `stl:content` expects a string as the value, `stl:repeat` expects a sequence. When this template is processed, the XML output will contain as many `<div>` elements as tasks are in the `tasks` variable. Within the div element, in each iteration over the `tasks` sequence, the variable `task` will be the respective item of the list.

The second new thing we see is the `<stl:block>` element. When the template is processed the `<stl:block>` tags are automatically removed.

Finally, look at the expression `task/id` or `task/title`, it shows the *slash* operator, which lets to traverse namespaces. So the variable `task` is expected to be a mapping (e.g. a dictionary), and `id` a key in that mapping, whose value is a string.

The Python side

Now let's see the Python code (see `examples/chapter9/TaskTracker.py`). Basically we have added the method `view` to the class `TaskTracker`:

```
def view(self):
    # Load the STL template
    handler = get_handler('TaskTracker_view.xml')
```

```
    # Build the namespace
    namespace = {}
    namespace['tasks'] = []
    for i, task in enumerate(self.state.tasks):
        namespace['tasks'].append({'id': i,
                                   'title': task.title,
                                   'description': task.description,
                                   'state': task.state,
                                   'is_open': task.state == 'open'})

    # Process the template and return the output
    return stl(handler, namespace)
```

To try the code run the Python interpreter and type:

```
>>> from itools.handlers import get_handler
>>> import TaskTracker
>>>
>>> task_tracker = get_handler('itools.tt')
>>> print task_tracker.view()
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:stl="http://xml.itools.org/namespaces/stl"
      xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h2>Task Tracker</h2>
    <div>
      <h4>
        #0:
        Re-write the chapter about writing handler classes.
        (<em>closed</em>)
      </h4>
      <p>A new chapter...
```

The Figure 14.2 shows how the HTML looks with a browser.

Task Tracker

#0: Re-write the chapter about writing handler classes. (*closed*)

A new chapter that explains how to write file handler classes must be written, it should go inmediately after the chapter that introduces file handlers.

#1: Finish the chapter about folder handlers. (*open*)

The chapter about folder handlers needs much more work. For example the skeleton of folder handlers must be explained.

Figure 14.2: The task tracker view

Chapter 15

TMX

Chapter 16

Uniform Resource Identifiers

In the wild Internet, the first challenge we encounter is how to identify and locate the numerous resources that populate it. Well, that's what **U**niform **R**esource **I**dentifiers (or just URIs) are for.

The Python Standard Library (batteries included!) provides a module named `urlparse` which is able to split a generic URI into its main parts (scheme, authority, path, query and fragment), to rebuild it, and to resolve relative references.

But there are more things we would like to do with a URI. For example we could go further in the parsing process and split the path into its segments, then split each segment into the name and the parameters if any; we could get the user information, host address and port number from the authority; we could normalize URIs; we could implement other operations beyond just resolving relative references, etc.

This is the purpose of `itools.uri`, to provide a complete API to parse and work with URIs, following the standard as described by **RFC2396**.

The main function provided by `itools.uri` is `get_reference`, a factory to build references:

`get_reference(reference)` - Parses the given string and returns a reference object.

Let's go right to the code:

```
>>> from itools import uri
>>> r1 = uri.get_reference('http://www.w3.org/TR/REC-xml/#sec-intro')
>>> r2 = uri.get_reference('mailto:jdavid@itaapy.com')
>>> r3 = uri.get_reference('http://www.ietf.org/rfc/rfc2616.txt')
>>> r4 = uri.get_reference('http://sf.net/cvs/?group_id=5470')
>>> r5 = uri.get_reference('news:comp.infosystems.www.servers.unix')
```

16.1 Syntax

Before going further, we will give an overview of the URI syntax, something required to understand the rest of this chapter.

A URI is divided in two parts. The first one is the scheme: `http`, `ftp`, `mailto`, etc.. The syntax and semantics of the second part depends on the scheme:

```
uri = <scheme>:<scheme-specific-part>
```

However, many schemes share a similar syntax for the second part, these URIs are known as *generic URIs*.

16.1.1 Generic URIs

The syntax of a generic URI reference is:

```
<scheme>://<authority><absolute path>?<query>#<fragment>
```

Generic URIs are modeled by `uri.Reference`. Following the code at the beginning, we are going to inspect the `r1` object:

```
>>> r1
<itools.uri.Reference object at 0x403ebc4c>
>>> print r1
http://www.w3.org/TR/REC-xml/#sec-intro
>>> print r1.scheme
http
>>> print r1.authority
www.w3.org
>>> print r1.path
/TR/REC-xml/
>>> print r1.query

>>> print r1.fragment
sec-intro
```

Note that there is an attribute for every component: the scheme, the authority, the path, the query and the fragment. Now we are going to quickly describe each of these components:

Scheme Defines the method or protocol to access the resource.

Authority Defines the server address that hosts the resource. Its syntax is:

```
authority = [<userinfo>@]<hostport>
```

Absolute path The path identifies the resource within the scope of the scheme and authority.

It consists of a sequence of segments. A segment has two parts, the name and the parameters, though the parameters are optional. The syntax is:

```
absolute path = /<relative path>
relative path = <segment>[/<relative path>]
segment = <name>[;<parameters>]
```

Query The query is information to be interpreted by the resource. It does not have a pre-defined syntax.

Fragment The fragment is a reference within the resource.

Actually, the fragment does not belong to the URI, as it does not help to identify the resource, however we include it here because it does appears in URI references, what is what we work with.

As the query, the fragment does not have a pre-defined syntax.

16.1.2 Non Generic URIs

Other schemes do not follow the generic syntax. As an example, let's inspect the `r2` object seen before:

```
>>> r2 = uri.get_reference('mailto:jdavid@itaapy.com')
>>> print r2
mailto:jdavi@itaapy.com
>>> r2
<itools.uri.Mailto object at 0x403f45ec>
>>> print r2.scheme
mailto
>>> print r2.username
jdavid
>>> print r2.host
itaapy.com
```

As you see the `r2` is not an instance of `uri.Reference`, but an instance of `uri.Mailto`.

16.2 Relative references

So far the examples we have seen show absolute URIs, but there are relative URI references too. A relative reference is one that lacks, at least, the scheme. There are three types of of relative references: network paths, absolute paths, and relative paths:

Network paths Network paths only lack the scheme, they start by a double slash and the authority, followed by the absolute path. They are rarely used.

```
//www.ietf.org/rfc/rfc2396.txt
```

Absolute paths The absolute paths lack both the scheme and the authority. They start by a slash.

```
/rfc/rfc2396.txt
```

Relative paths Relative paths lack the first slash of absolute paths. They can start by the special segment ".", or by one or more "..". Examples are:

```
rfc/rfc2396.txt
./rfc/rfc2396.txt
../rfc2616.txt
```

16.2.1 Resolving references

The most common operation with relative references is to resolve them. That is to say, to obtain (with the help of a base reference) the absolute reference that identifies our resource. This is achieved with the `resolve` method:

```
>>> base = uri.get_reference('http://www.ietf.org/rfc/rfc2615.txt')
>>> print base.resolve('//www.ietf.org/rfc/rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
>>> print base.resolve('/rfc/rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
>>> print base.resolve('rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
```

16.3 Paths

One component that deserves special attention is the path. The path of a generic URI is an instance of the `uri.Path` class:

```
>>> ref = uri.get_reference('http://www.ietf.org/rfc/rfc2616.txt')
>>> ref.path
<itools.uri.Path at 0x403f50a4>
>>> print ref.path
/rfc/rfc2616.txt
```

Paths are iterable:

```
>>> for segment in ref.path:
...     print segment
...
rfc
rfc2616.txt
```

Each component of the path is called a segment. Segments are instances of the class `uri.Segment`. Each segment has two components, the name and the parameter. The code below illustrates this:

```
>>> path = uri.Path('/itaapy;lang=es/team')
>>> for segment in path:
...     print repr(segment)
...     print '  name:', segment.name
...     print '  param:', segment.param
...
<itools.uri.Segment object at 0x404c1acc>
  name: itaapy
  param: lang=es
```

```
<itools.uri.Segment object at 0x404c1a2c>
  name: team
  param: None
```

The `uri.Path` class also provides an API to manipulate paths:

`is_absolute()`
- Returns `True` if the path is absolute (i.e. if it starts by an slash), `False` otherwise.

`is_relative()`
- Returns `True` if the path is relative (i.e. if it does not start by a slash), `False` otherwise.

`get_prefix(path)`
- Returns the path that is common to `self` and to the given path.

`resolve(path)`
- Returns a new path from a base path (`self`) and the given path. Follows the RFC2396 standard, i.e. takes into account the trailing slash.

`resolve2(path)`
- Same as `resolve`, but it does not teke into account the trailing slash.

`get_pathto(path)`
- Returns the path needed to go from `self` to the given path (this complements the `resolve` method).

`get_pathtoroot()`
- Returns a relative path to the root (something like `../../..`).

Chapter 17

Web

This chapter documents the high-level, cross-protocol, programming interface provided by `itools.web` to develop web applications.

17.1 The Publisher

The first and most important interface within the Web is the URI (Uniform Resource Identifier), both from the user's and programmer's point of view. This is how a URI looks like with `itools.web`:

```
http://localhost:8000/users/toto/;view
```

What is of most interest to us is the path, which expressed in a general form has the structure:

```
<path to resource>/;<method>
```

It is splitted in two parts, a path to a resource, and an action or view over that resource. The Figure 17.1 shows an excerpt of the application's structure, the excerpt that is relevant to our example.

So, the web application has a graph structure with a root node (an entry point to the graph). The URI path goes from the root to another node in the graph.

Once the node is reached, the given `<method>` of the resource is called. That's what the `itools.web` publisher does, follow the path to reach the node and call the method.

The use of the semicolon to separate the method from the path makes the URI explicit, you look at it and you know what is the path, which nodes in the graph the path goes through, and which method is called at the end.

Note that the character semicolon has been choosen because it is defined by the RFC 2396[1], which defines the URI standard. It is the character that, within a path segment, separates the resource name from the segment parameters.

[1] http://www.ietf.org/rfc/rfc2396.txt

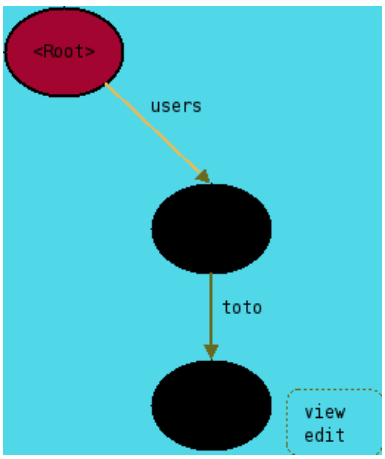

Figure 17.1: The application tree (an excerpt).

17.2 The Context

Wherever in the code there is available a global object named the *context*, it contains:

`request`
- The request object.

`response`
- The response object.

`user`
- The authenticated user, or `None` if it is an anonymous (non authenticated) user.

`root`
- The handler for the root resource.

`handler`
- The handler of the resource being published (usually it is `self`).

`path`
- The path from the root handler to the published handler (an instance of the `itools.uri.Path` class).

`method`
- The name of the method being published, or `None` if the url did not specified the method.

The context can be accessed through the `itools.web.get_context` method. A pattern that is very often found within the ikaaro's code is:

```
from itools.web import get_context

context = get_context()
request, response = context.request, context.response
```

17.2.1 The Request

The request object is a wrapper around the Zope's request object, it provides a higher level API:

`uri`
- The requested uri (an `itools.uri` reference).

`referer`
- The referer uri (an `itools.uri` reference), or `None` if there is not a referer.

`accept_language`
- An instance of the `itools.i18n.accept.AcceptLanguage` class, it keeps the user preferred languages.

`form`
- The request parameters passed either through the query or as form values. It is mapping from key to value.

`cookies`
- The cookies, it is a mapping.

17.2.2 The Response

The response object is a wrapper around the Zope's response object, it provides a higher level API:

```
has_header(name)
-
get_header(name)
-
set_header(value)
-
del_cookie(name)
-
set_cookie(name, value, **kw)
-
redirect(uri)
-
set_status(status)
-
```

Chapter 18

Workflow

Chapter 19

XHTML

We are going to give a glance to the implementation of XHTML provided by `itools.xhtml`.

To drive the explanation here is the document source we will use as example (see `examples/hello.xhtml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Hello world</title>
    <!-- Changed by: , 02-Jun-2004 -->
  </head>
  <body>
  </body>
</html>
```

The only two differences between this file and the example we saw at the beginning (`examples/chapter8/hello.xml`) are the *document type* declaration and the XML namespace declaration. The result is `get_handler` returns an XHTML document instead of an XML document:

```
>>> doc = get_handler('examples/hello.xhtml')
>>> doc
<itools.xhtml.XHTML.Document object at 0xb7a2ec4c>
>>>
>>> for node in doc.traverse():
...     print repr(node)
...
<itools.xhtml.XHTML.BlockElement object at 0xb79884ec>
u'\n  '
<itools.xhtml.XHTML.HeadElement object at 0xb798852c>
u'\n    \n    '
<itools.xhtml.XHTML.BlockElement object at 0xb79885ac>
u'Hello world'
u'\n    '
<itools.xml.XML.Comment object at 0xb798856c>
```

```
u'\n  '
u'\n  '
<itools.xhtml.XHTML.BlockElement object at 0xb798858c>
u'\n  '
u'\n'
```

Now, the element nodes are not any more instances of `XML.Element`, but instances of `XHTML.Element`, which extends the generic API with the methods:

`is_inline()` - Returns `True` if the element is an inline element, `False` otherwise.
`is_block()` - Returns `True` if the element is a block element, `False` otherwise.

Ok, not too much[1], but enough to give an idea of the power of `itools.xml`. In the next chapter we will see a much more compelling example of what can be done with `itools.xml`, the **S**imple **T**emplate **L**anguage.

[1] These two methods, `is_inline` and `is_block`, are actually really useful. They are used by the message extraction algorithm, a fundamental brick of the internationalization and localization services provided by `itools`.

Chapter 20

XLIFF

Chapter 21

eXtensible Markup Language (XML)

The purpose of this chapter is to explain the XML services provided by `itools`, which can be found in the sub-package `itools.xml`.

21.1 The parser

The first layer is the event driven parser implemented by `itools.xml.parser`, which is a wrapper around the `expat`[1] parser.

With `expat` you need a function for every event you want to manage, so for example, if you want to deal just with elements, comments and text nodes, you will need four functions (the start element, end element, comment and character data handlers), plus the main function that sets up the parser. State is typically shared across event handlers through instance variables. The `expat` approach makes it relatively hard to follow the program flow.

With `itools.xml.parser` all the code is within a single function, and state is stored in local variables. Let's see a dummy example:

```
from itools.xml import parser

for event, value, line_number in parser.parse(data):
    if event == parser.START_ELEMENT:
        namespace, local_name, attributes = value
        print 'START ELEMENT:', local_name
    elif event == parser.END_ELEMENT:
        namespace, local_name = value
        print 'END ELEMENT:', local_name
    elif event == parser.TEXT:
        print 'TEXT', value
```

The example above just prints a message to standard output each time the start of an element, the end of an element or a text node is found.

[1] http://expat.sourceforge.net/

The parser returns a list of events, where every event is a tuple of three values: the event type, the value (which depends on the event type) and the line number. The events implemented are:

Event	Value
`DOCUMENT_TYPE`	(name, system id, public id, has internal subset)
`START_ELEMENT`	(namespace uri, local name, attributes)
`END_ELEMENT`	(namespace uri, local name)
`COMMENT`	value
`TEXT`	value
`NAMESPACE`	namespace uri

For the `START_ELEMENT` event, the attributes are passed as a dictionary where the key is a tuple of the namespace URI and the local name of the attribute, and the value is the value of the attribute. For example:

```
{('http://www.w3.org/1999/xhtml', 'src'): <itools.uri.generic.Reference>,
 ('http://www.w3.org/1999/xhtml', 'title'): u'Google'}
```

Note that the attribute value may not be an string, as the example above illustrates. Within `itools.xml` we use namespaces not only to check correctness of the XML document, but also to de-serialize values and build a higher level data structure.

Note also that the parser does not returns the prefixes used within the document, and there is not an event for the XML declaration either. We consider this data to be internal to the document, it does not reveal any information about the document's logical structure, hence it is not valuable for our purposes.

21.2 Namespaces

From the table above you can see `itools.xml` provides support for XML namespaces. The event values for elements and attributes are tuples, where the first two components are the *namespace uri* and the *prefix*. If an element or attribute is not attached to a namespace, the *uri* and the *prefix* will be `None`.

The module `itools.xml.namespaces` provides a registry for namespace handlers, and an abstract class which defines the programming interface and provides a default behaviour for subclasses. The table below defines this programming interface:

`class_uri`
- The uri that uniquely identifies this namespace.

`class_prefix`
- The recommended prefix for the namespace. For example "`dc`" for Dublin Core. By default it is `None`.

`get_element_schema(name)`
- Returns the schema for the given element name.

`get_attribute_schema(name)`
- Returns the schema for the given attribute name.

The schema is a dictionary whose keys and values are somewhat arbitrary, they will depend on what you need. Though, it usually includes the type of the attribute or element, which is used to deserialize and serialize the values. For example, consider a link within an XHTML document, like:

```
<a href="http://www.example.com" title="Example" />
```

The `href` attribute should be loaded as a URI reference, and `title` should be a unicode string. The excerpt below will do the job:

```
from itools.xml import parser, namespaces
from itools import xhtml

for event, value, line_number in parser.parse(data):
    if event == parser.ATTRIBUTE:
        namespace_uri, prefix, local_name, value = value
        namespace = namespaces.get_namespace(namespace_uri)
        schema = namespace.get_attribute_schema(local_name)
        value = schema['type'].decode(value)
        print local_name, schema['type']
        print repr(value)
        print
```

The output when running this code is:

```
href <class 'itools.handlers.IO.URI'>
<itools.uri.generic.Reference object at 0xb7a8c8ac>

title <class 'itools.handlers.IO.Unicode'>
u'Example'
```

For examples about how to define your own namespace handlers see the Dublin Core (`itools.xml.DublinCore`) and XHTML (`itools.xhtml.XHTML`) implementations.

21.3 Documents

If `itools.xml.parser` provides an event driven parser, whose function is similar to SAX[2], we do have too an equivalent for DOM[3]. And of course it takes the form of a resource handler.

The handler class `itools.xml.XML.Document` loads the document into memory as a tree of nodes, with some global attributes. Let's inspect an example:

```
>>> from itools.handlers import get_handler
>>> import itools.xml
>>>
>>> doc = get_handler('examples/chapter8/hello.xml')
>>> doc
<itools.xml.XML.Document object at 0x4064466c>
>>> print doc.xml_version
```

[2] http://XXX
[3] http://XXX

```
1.0
>>> print doc.standalone
-1
>>> print doc.document_type
None
>>> print doc.root_element
<itools.xml.XML.Element object at 0xb7a3272c>
```

The code above shows the four attributes that keep the document's state:

`xml_version`
- The XML version of the document, usually it is `1.0`.

`standalone`
- Possible values are: `1` if the document was declared standalone, `0` if it was declared not to be standalone, or `-1` if the standalone clause was omitted.

`document_type`
- If the document lacks a document type declaration this attribute will be `None`. If the document type was specified this attribute will be a tuple with four values: the name, the system id, the public id and a boolean that tells wether the document contains an internal declaration subset.

`root_element`
- An instance of the `XML.Element` class, the root of the DOM-like tree that represents the XML data, and that we will study later with more detail.

The API for the documents is rather simple:

`get_root_element()`
- Returns the root element.

`traverse()`
- A generator that traverses the XML tree in pre-order, and returns each time a node. It is a shorthand for `root_element.traverse()`.

`traverse2()`
- A more powerful version of `traverse`. It is a shorthand for `root_element.traverse2()`.

21.3.1 Inspecting the tree

Coming back to the example, the `examples/hello.xml` file's content is:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Hello world</title>
    <!-- Changed by: , 02-Jun-2004 -->
  </head>
  <body>
```

```
  </body>
</html>
```

The method `traverse` lets us to easily inspect the tree nodes:

```
>>> for node in doc.traverse():
...     print repr(node)
...
<itools.xml.XML.Element object at 0xb7a3316c>
u'\n  '
<itools.xml.XML.Element object at 0xb7a331ac>
u'\n    '
<itools.xml.XML.Element object at 0xb7a332ac>
u'\n    '
<itools.xml.XML.Element object at 0xb7a331ec>
u'Hello world'
u'\n    '
<itools.xml.XML.Comment object at 0xb7a335cc>
u'\n  '
u'\n  '
<itools.xml.XML.Element object at 0xb798574c>
u'\n  '
u'\n'
```

Here we see the three kind of nodes currently supported: elements, comments and text nodes.

21.3.2 Elements

Of the three kinds of nodes, elements are the most important and complex. Element nodes give the tree structure, as they are the only ones that may have children. If you inspect an element you will see the following attributes:

`namespace`
- The XML namespace of the element (it will be None for a bare element).

`prefix`
- The prefix used for the element's XML namespace.

`name`
- The name of the element.

`attributes`
- A dictionary mapping from the tuple XML namespace and local name to the attributes value.

`prefixes`
- A mapping from XML namespace to prefix (used to get the qualified name of an attribute).

`children`
- A list with the children of the element (elements, comments and text nodes).

And here is the API:

`get_qname()`
- Returns the qualified name of the element.

`copy()`
- Returns a clone of the element.

`set_attribute(namespace, local_name, value, prefix=None)`
- Sets the given attribute.

`get_attribute(namespace, local_name)`
- Returns the attribute value for the given XML namespace and local name.

`has_attribute(namespace, local_name)`
- Returns a boolean value, true if the element contains a value for the given XML namespace and local name, false otherwise.

`get_attributes()` - Is a generator that returns a three value tuple each time, the values are: XML namespace uri, local name, value.

`get_attribute_qname(namespace, local_name)`
- Returns the qualified name for the given XML namespace and local name.

`set_element(element)`
- Appends the given element to the list of children.

`set_comment(comment)`
- Appends the given comment to the list of children.

`set_text(text)`
- Appends the given text node (a unicode value) to the list of children.

`get_elements(name=None)`
- Returns a list with all the element nodes whose name is the given name; if the parameter `name` is not given, then return all the element nodes.

`traverse()`
- A generator that traverses the element's children in pre-order, and returns each time a node.

`traverse2()`
- A more powerful version of `traverse`.

Appendix A

Coding style guide

This chapter describes the coding conventions used to write `itools`. If you ever contribute a patch to `itools`, be sure your code adheres to these rules.

Sometimes this guide contradicts the *PEP 8*[1] recommendations; in these cases this guide applies, in the context of `itools`.

A.1 Language and encoding

Code must be written in english. The preferred encoding is *UTF-8*, though others encodings are allowed.

A.2 Module structure

Each module is splitted in six sections:

1. encoding statement;
2. copyright notice;
3. license reference;
4. the module documentation string;
5. import statements;
6. the code itself.

Here we are going to describe the module's header, made up of the first five sections. The style for the code itself will described in the rest of this chapter.

The encoding

It is only requiered if it is any other encoding than ASCII. Example:

```
# -*- coding: UTF-8 -*-
```

[1] http://www.python.org/peps/pep-0008.html

The Copyright

After the encoding comes the copyright, whose structure is:

```
# Copyright (C) <years> <author name> <email>
                <years> <author name> <email>
                ...
```

The License

Right after the copyright statement comes a reference to the license. For `itools` it is the *LGPL*.

The module's documentation string

There should be a documentation string explaining what the module does (though it is better to have none than a dummy one).

The explanation about how to write a documentation string is out of the scope of this chapter, it is covered by the *PEP 257*[2].

Imports

The imports statements are at the top of the file (just after the legal statements and the module docstrings), though there may be imports within a function or method to avoid circular references.

Imports should be grouped, with the order being:

1. standard library imports
2. `itools` imports
3. other package package imports

Every group should be preceded by an introductory comment, like:

```
# Import from the Standard Library
...

# Import from itools
...
```

A.2.1 Example

For example, at the time of this writting the module `itools.handlers.Handler` starts by:

```
# -*- coding: ISO-8859-1 -*-
# Copyright (C) 2003-2004 Juan David Ibáñez Palomar <jdavid@itaapy.com>
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 2.1 of the License, or (at your option) any later version.
```

[2] http://www.python.org/peps/pep-0257.html

```
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA

"""
This module provides the abstract class which is the root in the
handler class hierarchy.
"""

# Import from the Standard Library
import datetime

# Import from itools
from itools import uri
from itools.resources import base
```

A.3 Format rules

Indentation: 4 spaces

Each indentation level must have four (4) spaces. Never use tabs.

Maximum line length: 80 characters

Lines should be 80 characters wide at most.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but usually using a backslash looks better. Make sure to indent the continued line appropriately.

One line, one statement

Don't put more than one statement on the same line:

	Good	Bad
1	`if x is True:` `  do_something()`	`if x is True:  do_something()`

2	`do_one()`
	`do_two()`
	`do_three()`
		`do_one(); do_two(); do_three()`

3	`def get_area(x, y):` `  return x * y`	`def get_area(x, y):  return x * y`

Blank lines

Separate classes with three blank lines. Separate methods and functions with two blank lines. There is also a blank line between the class definition and the first method definition.

Use blank lines in functions, sparingly, to indicate logical sections.

Whitespace in expressions and statements

Surround operators with one white space. Arithmetics operators may be the exception, for them it is possible to use either one space or none, whatever reads better. But never use more than one space.

Well, there is another exception, the sign "=" used in keyword arguments should not be surrounded by spaces. Type `Document(title="hello")` instead of `Document(title = "hello")`.

Never add spaces neither before nor after parentheses, brackets or braces. The only exception is for list comprehensions, where it is allowed to add a space after the opening bracket, and another space before the closing bracket.

The comma and colon must be followed by a space (or a new line), but never put a space before. The only exception is for one element tuples, where the comma must be immediately followed by the closing parentheses. The semicolon should never be used.

A.4 Comments

Comments must describe the code that follows them, and must be indented to the same level of that code. Inline comments are not allowed; this is to say, a comment always starts a new line.

A comment starts by a single `#` character followed by a space.

Comments must be written in good english (as good as the developer can write it). This means, for example, that the first letter must be capitalized.

A.5 Naming conventions

The names of *variables*, *classes*, *functions*, *methods* and *constants* are written with one or more english words. Most of the words used are *nouns*, *verbs*, and *adjectives*.

Abbreviations may be used, but in general it is preferred the complete word, for example, `language` instead of `lang`. When an abbreviation is not obvious, its meaning should be explained with a comment.

The allowed naming conventions are three:

lower_case_with_underscores All words are in lowercase and separated by an underscore. This convention is used for *variables*, *functions* and *methods*.

UPPER_CASE_WITH_UNDERSCORES All words are in uppercase and separated by an underscore. Used only for *constants*.

CapitalizedWords All words start by an uppercase, with the rest of the word in lowercase. Words are not separated by any character, the uppercase

letters serve to visually distinguish when a new word starts. Used only for *classes*.

A.5.1 Class names

Class names are written in capitalized words. Typically they are made of nouns and/or adjectives.

A.5.2 Functions and methods

Functions and methods are written in lowercase with underscores.

They must start by a verb, and they should be followed by a complenent that clearifies what the funciont does. For example, it is better to spell `set_object` than just `set`.

A.5.3 Variables

Variables are written in lowercase with underscores. Most of the time they are nouns with or without adjectives.

One letter variables may be used in mathematical expressions, for sequence indexes, or in comprehensive lists:

```
>>> public_handlers = [x for x in handlers if x.state == 'public']
```

A.5.4 Constants

Constants are written in uppercase with underscores.

Appendix B

GNU arch

A control version system is a tool that keeps track of changes made in the code, when a change was made, by whom. It helps multiple developers to work on the same project, to merge the different changes they made in the same code base.

GNU arch[1] (also known as *tla*) is a modern and advanced control version system. It is the one we use to manage the `itools` source code.

There are many ways to work with *tla*, this appendix explains the one we use for `itools`.

The exposition is organized in three sections that detail:

1. How to keep track of the development of `itools`.
2. How to maintain private changes.
3. How to contribute your changes back to the main development tree.

B.1 Keeping track of `itools`

You may want to have the last bleeding edge features from `itools` in your system as soon as they are written, or to track how the development is going on. Then this section is for you.

B.1.1 Browsing the sources

To browse the `itools` archive tree through the web, just go the url below:

```
http://in-girum.net/cgi-bin/archzoom.cgi/jdavid@itaapy.com--public
```

B.1.2 Check out

To check out `itools` from the archive you need to install *tla*. Most distributions include it, for example, if you use Gentoo[2] just type:

```
$ sudo emerge tla
```

Once *tla* is installed, follow the steps described below.

[1] `http://www.gnu.org/software/gnu-arch/`
[2] http://www.gentoo.org

Set your id

```
$ tla my-id "Toto Bonaparte <toto@example.com>"
$ tla my-id
Toto Bonaparte <toto@example.com>
```

Register the official `itools` archive

```
$ tla register-archive jdavid@itaapy.com--public \
      http://in-girum.net/~jdavid/archives/public
$ tla archives
jdavid@itaapy.com--public
    http://in-girum.net/~jdavid/archives/public
```

Check out `itools`

```
$ cd ~/sandboxes
$ tla get jdavid@itaapy.com--public/itools--main--0.6 itools-0.6
$ cd itools-0.6
$ tla tree-version
jdavid@itaapy.com--public/itools--main--0.6
```

B.1.3 A session with *tla* and `itools`

Now, whenever you want to see if something has changed in `itools`, just type:

```
$ cd ~/sandboxes/itools-0.6
$ tla missing --summary
patch-80
    use Python's documentation to profile the catalog
patch-81
    fix XML error handling (hence better STL message errors)
```

The output shows the new patches available (if your code is up-to-date the output will be empty). Say you want to apply the patches, type:

```
$ tla update
[...]
```

B.1.4 Help

The Table B.1 summarizes the *tla* commands seen in this section. To learn about other commands use `tla help`, and for details about a command type:

```
$ tla <command> --help
```

B.2 Maintaining private changes

Now maybe you want to make some changes to `itools`. The wisest to do in this situation is to create a branch of `itools`, this will let you to easily update to the last version while keeping your changes.

The first step is to setup an archive (if you have already one you can skip to the next subsection).

`tla my-id` - Print or change your id.
`tla register-archive` - Change an archive location registration.
`tla archives` - Report registered archives and their locations.
`tla get` - Construct a project tree for a revision.
`tla tree-version` - Print the default version for a project tree.
`tla missing` - Print patches missing from a project tree.
`tla update` - Update a project tree to reflect recent archived changes.
`tla help` - Provide help with arch.

Table B.1: Basic *tla* commands

B.2.1 Create an archive

```
$ mkdir ~/{archives}
$ mkdir ~/{archives}/public
$ tla make-archive toto@example.com--public ~/{archives}/public
$ tla archives
jdavid@itaapy.com--public
    http://in-girum.net/~jdavid/archives/public
toto@example.com--public
    /home/toto/{archives}/public
```

Make it your default archive:

```
$ tla my-default-archive toto@example.com--public
$ tla my-default-archive
toto@example.com--public
```

B.2.2 Create a branch

With your own archive, it is time to create a branch of `itools`:

```
$ tla tag -S jdavid@itaapy.com--public/itools--main--0.6 itools--toto--0.6
  * creating category toto@example.com--public/itools
  * creating branch toto@example.com--public/itools--toto
  * creating version toto@example.com--public/itools--toto--0.6
  * Archive caching revision
```

So now you can replace the check-out from the main tree with a one from your own branch:

```
$ cd ~/sandboxes
$ rm -rf itools-0.6
$ tla get toto@example.com--public/itools--toto--0.6 itools-0.6
$ cd itools-0.6
$ tla tree-version
toto@example.com--public/itools--toto--0.6
```

B.2.3 Working with your branch

So, now you modify `itools` to add a new feature. Every change made to a file will be automatically detected by *tla*, but if your work includes new files or directories, or you have removed, renamed or moved a file or a directory, then you must tell *tla* about these changes, to do so use the commands below:

```
$ tla add <filename>
[...]
$ tla delete <filename>
[...]
$ tla move <old filename> <new filename>
[...]
```

Maybe you forgot to add a file, before committing is a very good idea to verify it with the command `tree-lint`:

```
$ tla tree-lint
[...]
```

This command looks at your project tree and tells you about files suspected to be source code that have non been added yet.

Ok, so you have finished working on this new cool feature and are willing to check it in your branch of `itools`. First, verify what you have changed:

```
$ tla changes
[...]
```

This command shows which files (and folders) have been modified, removed, added or moved. For a more detailed description, try:

```
$ tla changes --diffs | less
```

Take your time to examine the changes, maybe you forgot to remove a print statement? a close look at the output of `tla changes --diffs` will tell you.

Once you are sure everything is alright, it came the time to commit. First you have to write a log message:

```
$ vi ‘tla make-log‘
```

Within the editor, you should introduce a title that describes the changes you have done, and optionally a longer description. Once you are done, left the editor and type:

```
$ tla commit
$ tla revisions
[...]
patch-1
    add feature XXX
```

B.2.4 Merging from the main branch

Ok, so now the upstream version of `itools` is modified, how to merge the changes in your tree? easy:

```
$ cd ~/sandboxes/itools-0.6
$ tla star-merge -t jdavid@itaapy.com--public/itools--main--0.6
[...]
```

Beware, there may be conflicts that you must resolve.

Now, your project tree contains the changes from the upstream archive, you must commit them in your own archive. The log is written automatically by typing:

```
$ tla log-for-merge >> ‘tla make-log‘
$ vi ‘tla make-log‘
```

Within the editor there will be a description detailing the patchs that have been applied. So you just have to add the subject, something like “merging from the main tree”. Once this is done just commit as usual:

```
$ tla commit
```

Summary

See the Table B.2 for a summary of the *tla* commands seen in this section.

B.3 Contributing your work to the main tree

To contribute your changes back to the main development branch you must make your branch available through internet. We assume the archive you have set-up is your local computer, so you have to create a mirror of your archive from your local computer to an internet server:

```
$ tla make-archive --listing --mirror toto@example.com--public \
      sftp://example.com/home/toto/{archives}/public
```

You must be sure your mirror can be accessed through an internet protocol, like HTTP or FTP. So other developers will be able to merge from your branch.

Now, whenever you make and commit a change, to share that change with the community, you have to synchronize your mirror:

```
$ tla archive-mirror
[...]
```

See the Table B.3 for a summary of the commands seen in this section.

`tla make-archive` - Create a new archive directory.
`tla my-default-archive` - Print or change your default archive.
`tla tag` - Create a continuation revision (aka tag or branch).
`tla add` - Add an explicit inventory id.
`tla delete` - Remove an explicit inventory id.
`tla move` - Move an explicit inventory id.
`tla tree-lint` - Audit a source tree.
`tla changes` - Report about local changes in a project tree.
`tla make-log` - Initialize a new log file entry.
`tla commit` - Archive a changeset-based revision.
`tla revisions` - List the revisions in an archive version.
`tla star-merge` - Merge mutually merged branches.
`tla log-for-merge` - Generate a log entry body for a merge.

Table B.2: Maintaining private changes: summary

`tla archive-mirror` - Update an archive mirror.

Table B.3: Maintaining private changes: summary

Index