

GNUe Common: A Developer's Introduction

A Guide to Developing Applications with GNUe Common

GNUe-Common Version 0.5.15

Last Revised 05/06/05 07.25.39 pm GMT+1

Copyright 2000-2005 Free Software Foundation

Written by James Thompson

Revised and updated by Davor Ocelić, docelic@mail.inet.hr

Table of Contents

1. Introduction.....	2
2. A Sample GNUe Application.....	3
The Initial Code.....	3
Adding Command Line Option Support.....	5
Adding Debugging Output Support.....	6
Adding Configuration File Support.....	7
3. Using GNUe DataSources.....	10
Introduction.....	10
4. Using GParser	11
Introduction.....	11
5. Using Triggers.....	12
Introduction.....	12
6. Using Events.....	13
Introduction.....	13
XX. A Complete GNUe-Common App.....	14
Appendix A: Schema Definition Elements.....	15
Schema Tags.....	15
Data Tags.....	15
Tables Tags.....	16
Import Tags.....	18
Appendix B: Datasource Drivers.....	21
Supported Databases.....	21
Detailed Middleware Listings.....	30

1. Introduction

The GNUe Common Library is a set of classes and utilities designed to provide a large amount of basic application functionality with a minimal amount of effort on the part of the application developer. This library serves as the core for the GNUe tools, such as Forms, Reports, Application Server, and Designer.

Some of the key features include.

- A database-abstraction layer that provides support for most major databases. Database sources can be mixed and matched transparently in many cases. Changing the source of data typically involves changing 1 or 2 lines in a single configuration file.
- A builtin XML-to-Object parser and Object-to-XML marshaller. You can rapidly define xml based file formats via python dictionaries.
- An RPC abstraction layer that will allow server processes to define their public methods once and have them available to CORBA, XML-RPC, SOAP, and DCOM clients.
- A configuration system system that provides system-wide configurations as well as per user adjustments. System wide configuration entries can optionally be locked. Command line options are also easily added to the system.
- A trigger system that allows the application developer to provide hooks into their systems to allow easier customization using python methods. Triggers are provided an easily extended, customized namespace that includes hooks to objects in memory, custom functions embedded in the primary app.
- Integrated debugging features including the integrated python debugger, profiler, debug levels, log files. Access to these features requires little to no coding on the developers part.
- A flexible internal event system to allow an application's sub systems to register for events and react upon them.
- Automatic generation of various levels of documentation. This documentation is pulled directly from the internal structures defined in your application and includes configuration options, run-time command line options, manual pages, and xml file descriptions.

2. A Sample GNUE Application

No tutorial for a programming environment would be completely without a functional Hello World application. In this chapter, we will create our hello world application. We will then extend that application to use a few of the more basic GNUE-Common features.

The Initial Code

Lets take a look at our complete helloworld application. Create a file named helloworld with the following contents.

```
#
#Import the base application support
#
from gnue.common.apps.GClientApp import *

#
# Define our application
#
class Hello(GClientApp):
    VERSION = "0.0.1"
    COMMAND = "helloworld"
    NAME = "Hello World"
    USAGE = GClientApp.USAGE
    SUMMARY = _("App to display the text Hello World .")
    AUTHOR = "GNU Enterprise Project"
    EMAIL = "info@gnue.org"
    REPORT_BUGS_TO = "Report bugs to info@gnue.org."

    def run(self):
        print "Hello World!"

if __name__ == '__main__':
    Hello().run()
```

After you have created your helloworld file, proceed to execute it.

```
bash-2.05a$ python helloworld
Hello World!
bash-2.05a$
```

That isn't very much output for such a large amount of code. What's going on here? Let's rerun the application using the following command.

```
bash-2.05a$ python helloworld --help
Hello World
Version 0.0.1

GNUE Common Version 0.4.1a

Usage: helloworld [options]

App to display the text Hello World .

Available command line options:

  --configuration-options
                        Displays a list of valid configuration file entries,
                        their purpose, and their default values.

  --connections <loc>
                        Specifies the location of the connection definition
                        file. <loc> may specify a file name
                        (/usr/local/gnue/etc/connections.conf), or a URL
                        location (http://localhost/connections.conf). If this
                        option is not specified, the environent variable
                        GNUE_CONNECTIONS is checked.

  --debug-file <file>
                        Sends all debugging messages to a specified file
                        (e.g., "--debug-file trace.log" sends all output to
                        "trace.log")
```

```

--debug-level <level>      Enables debugging messages. Argument specifies the
                             level of messages to display (e.g., "--debug-level 5"
                             displays all debugging messages at level 5 or below.)

--generate-man-page        Generates a groff-formatted man page as a file in the
                             current directory.

--help                      Displays this help screen.

--interactive-debugger      Run the app inside Python's built-in debugger

--profile                  Run Python's built-in profiler and display the
                             resulting run statistics.

--version                   Displays the version information for this program.

Please report any bugs to info@gnue.org.
bash-2.05a$

```

Do all those options really work? Yes. Our helloworld application supports an integrated debugger, profiler, and a fair bit of self documentation. Now we will look at the various sections of code and explain their function.

```

#
# Import the base application support
#
from gnue.common.apps.GClientApp import *

#
# Define our application
#
class Hello(GClientApp):

```

GNUe Common provides two different classes, GClientApp or GserverApp. These classes, while optional, provide a great deal of functionality to applications that are based upon them. Almost all official GNUe Applications are based upon one of these two classes. GClientApp should be used when your application will not be required to run as a daemon process. If you have such a requirement then you will use GserverApp. Our helloworld application is based upon GClientApp.

```

VERSION = "0.0.1"
COMMAND = "helloworld"
NAME = "Hello World"
USAGE = GClientApp.USAGE
SUMMARY = _("App to display the text Hello World .")
AUTHOR = "GNU Enterprise Project"
EMAIL = "info@gnue.org"
REPORT_BUGS_TO = "Report bugs to info@gnue.org."

```

These lines are used to set default values displayed on various help screens and documentation formats. Looking closer at the SUMMARY definition you will notice the text string is surrounded by `_()`. GNUe Common contains full support for `i18n`¹. For now we will not be examining that support, it will be saved for a later chapter.

```

def run(self):
    print "Hello World!"

if __name__ == '__main__':
    Hello().run()

```

The actual application code.

¹ Internationalization

Adding Command Line Option Support

Our helloworld is a success but it seem seems to be missing something. GNUE developers are scattered all over the globe, but helloworld only displays it's message english. Lets make a few changes to the code to add the following features

- The word *world* will be replaced with any name that is passed in it on the command line.
- An `-w(--welcome)` option will be added to toggle on the printing of the word Welcome
- An `-L(--language)` option will be added to allow the user to choose the language in which we display our welcome.² We will support english and Maori as our valid languages.

Here is the new code with the bolded sections containing the altered code. In all our examples, it is important that you do not accidentally insert TAB characters or misalign the lines, or you'll run into compile-time errors.

```
#
#Import the base application support
#
from gnue.common.apps.GClientApp import *

#
# Define our application
#
class Hello(GClientApp):
    VERSION = "0.0.1"
    COMMAND = "helloworld"
    NAME = "Hello World"
    USAGE = GClientApp.USAGE + ' [name]'
    SUMMARY = _("App to display the text Hello World .")
    AUTHOR = "GNU Enterprise Project"
    EMAIL = "info@gnue.org"
    REPORT_BUGS_TO = "Report bugs to info@gnue.org."

    def __init__(self):
        self.addCommandOption ('welcome_option', 'w', 'welcome',
                               help = _("Display the welcome message.))

        self.addCommandOption ('lang_option', 'L', 'lang',
                               argument = 'language',
                               default = 'english',
                               help = _("The language to print welcome "
                                       "Valid values: english, maori"))

        GClientApp.__init__(self)

    def run(self):
        greetings = { 'english' : 'Welcome',
                     'maori' : 'Kia Ora'
                     }

        if self.ARGUMENTS:
            print "Hello %s!" % self.ARGUMENTS[0]
        else:
            print "Hello World!"

        if self.OPTIONS['welcome_option']:
            print greetings[self.OPTIONS['lang_option']]

if __name__ == '__main__':
    Hello().run()
```

Let's rerun the application using the following command.

```
| bash-2.05a$ python helloworld --help
```

You will notice a few more options listed in the help screen and on man pages generated via the `-generate-man-page` option.

² True i18n support would be too complicated for our simple example.

```

--lang <language>, -L          The language to use to print thank youValid values:
                                english, maori
--welcome, -w                  Display the welcome

```

Now lets try running helloworld with a few different options.

```

bash-2.05a$ python helloworld.py
Hello World!
bash-2.05a$ python helloworld.py -w
Hello World!
Welcome
bash-2.05a$ python helloworld.py -w -L maori
Hello World!
Kia Ora
bash-2.05a$ python helloworld.py -w -L maori andrew
Hello andrew!
Kia Ora
bash-2.05a$

```

The key to adding options is the code

```

self.addCommandOption ('welcome_option', 'w', 'welcome',
    help = _("Display the welcome message.))

self.addCommandOption ('lang_option', 'L', 'lang',
    argument = 'language',
    default = 'english',
    help = _("The language to print welcome "
        "Valid values: english, maori"))

```

addCommandOption is a function that adds entries that to your application's list of valid options. The function takes the following parameters:

(self, name, shortOption, longOption, acceptsArgument, default, argumentName, help, category, action, argument)

Argument	Description
self	self
name	The key name that will be available in the self.OPTION dictionary when the application is executing.
shortOption	The single letter to assign to this option
longOption	The long option name. The is prepended with -- on the command line.
acceptsArgument	True if the option requires a value to be assigned from the command line.
default	This is the default value if the option is not passed in via the command line.
argumentName	Same as argument. argument overrides argumentName.
help	The description of the option displayed in help text.
category	used to create groups of command options, where there are the groups "base", "dev", "connections" and "general" predefined. There is an option --help-dev, --help-connections to give a special help-text for these groups of options.

action	A function-pointer; if supplied this function will be called automatically if the option is given on command line.
argument	This is used when generating the outputting help text. In our example above you will notice --lang <language>. The word 'language' inside the <> is set by this value. Same as argumentName

Adding Debugging Output Support

It is often handy to track what is going on inside a running copy of a program. You could start helloworld using the --interactive-debugger option and be dropped directly into python's debugger. This is a powerfull way to track what is happening inside your application but it can also be inconvenient for simple debugging tasks. It is also a bit too much to ask of an end user that is requiring technical assistance..

Let's tweak our program a bit. In the interest of saving trees we'll only show small pieces of code with the new parts in bold.

```
#
#Import the base application support
#
from gnue.common.apps.GClientApp import *
from gnue.common.apps import GDebug
```

Here we load GNUe Common's Gdebug system.

```
if self.ARGUMENTS:
    gDebug(5,'The value to be printed is %s' % self.ARGUMENTS[0])
    print "Hello %s!" % self.ARGUMENTS[0]
else:
    gDebug(5,'The default value will be printed')
    print "Hello World!"
```

gDebug() will output to either a screen or a file depending upon the options passed to the program at start. The first argument specifies the debug level (or level range) required before the debug text will print. The 5 in the example means that --debug-level 5 must be specified on the command line. The GNUe project typically uses these values:

- 1: Warnings (things that shouldn't happen when everything was perfect) (e.g. depreciation warning, unicode conversion warning etc.)
- 2: info about system environment (python version, command line args, environment variables)
- 3: SQL commands sent to the backend, or similar information for non-SQL backends
- 4-6: can be used by the tools like forms, reports, appserver
- 7-9: can be used by common

To see different level debug messages from the GNUe framework, try running the program with --debug-level 2, --debug-level 1,3 or --debug-level 1-10.

Adding Configuration File Support

It is often nice to allow application options to be set at various levels. GNUe Common supports a configuration file system with the following features

- Application default settings in the code.

- A systemwide configuration file that can be overridden by individual user configuration files.
- A systemwide configuration file that can not be overridden by individual user configuration files.
- Auto-documentation support. All GNUe Common apps support a command line option (--configuration-options) that displays all valid configuration file options and their default values.

We will now alter our helloworld application to allow users to replace the world Hello with the text of their choice. Rather than continue to print only pieces of the code we will again display a complete copy of our application. All the current changes are in bold.

```

#
#Import the base application support
#
from gnue.common.apps.GClientApp import *
from gnue.common.apps import GDebug
from gnue.common.formatting import GTypecast

#
# Define our application
#
class Hello(GClientApp):
    VERSION = "0.0.1"
    COMMAND = "helloworld"
    NAME = "Hello World"
    USAGE = GClientApp.USAGE + ' [name]'
    SUMMARY = _("App to display the text Hello World .")
    AUTHOR = "GNU Enterprise Project"
    EMAIL = "info@gnue.org"
    REPORT_BUGS_TO = "Report bugs to info@gnue.org."
    CONFIGFILE="ourapp.conf"

    def __init__ (self):

        self.addCommandOption ('welcome_option', 'w', 'welcome',
            help = _("Display the welcome message. "))

        self.addCommandOption ('lang_option', 'L', 'lang',
            argument = 'language',
            default = 'english',
            help = _("The language to print welcome "
                "Valid values: english, maori"))

        ConfigOptions = (
            { 'Name'      : 'greetingText',
              'Type'      : 'Setting',
              'Comment'   : 'Use the basic editor for triggers',
              'Description': 'Use the basic editor for triggers',
              'Typecast'  : GTypecast.text,
              'Default'   : 'Hello' },
            )

        GClientApp.__init__ (self, application="helloworld",
            defaults=ConfigOptions)

    def run(self):
        greetings = { 'english' : 'Welcome',
                     'maori'   : 'Kia Ora'
                    }

        if self.ARGUMENTS:
            gDebug(5, 'The value to be printed is %s' %
                self.ARGUMENTS[0])
            print "%s %s!" % (gConfig('greetingText'),self.ARGUMENTS[0])
        else:
            gDebug(5, 'The default value will be printed')

        if self.OPTIONS['welcome_option']:
            print greetings[self.OPTIONS['lang_option']]

if __name__ == '__main__':
    Hello().run()

```

Lets look at the various sections of added code.

```
from gnue.common.formatting import GTypecast
```

This imports a basic type casts library that we'll be using to define the types of configuration values we'll expect.

```
ConfigOptions = (
    { 'Name'      : 'greetingText',
      'Type'      : 'Setting',
      'Comment'   : 'Use the basic editor for triggers',
      'Description': 'Use the basic editor for triggers',
      'Typecast'  : GTypecast.text,
      'Default'   : 'Hello' },
)
```

Here we are defining a python dictionary of valid configuration file options. In this case we only have 1 option named `greetingText`, it is of type `text`, and defaults to `hello`. The type and comment values are currently unused. The description value is use when generating help text with the user runs the application with `-configuration-options`.

```
GClientApp.__init__(self, application="helloworld",
                   defaults=ConfigOptions)
```

Now we initialize our `GClientApp` parent, passing in the configuration options via the `defaults=ConfigOptions`. By doing this the application will do three things.

1. It loads the default value defined in the `greetingText` definition.
2. It will scan for a `/usr/local/gnue/etc/ourapp.conf` file.
 - It will load the file and look in the `[DEFAULT]` section for a value of `'greetingText'`, if found it will store the value.
 - It will then look for a `[helloworld]` section in the same file. If found it will replace any previous value stored for `greetingText`
3. It will next scan for a `$HOME/.gnue/ourapp.conf` file. If found it will perform the same scans as it did on the system scans in step 1.
4. Finally it will look for a `/usr/local/gnue/etc/ourapp.conf.fixed`. If found it will perform the same scans as it did on the system scans in step 1 and 2. Since this file is scanned last it has the effect of locking any settings the site admin has decided to prevent users from overriding.

```
print "%s %s!" % (gConfig('greetingText'),self.ARGUMENTS[0])
```

Here we use the global function `gConfig()` to pull the derived value of `greetingText` from the configuration system. Now lets test. We will need to create the `ourapp.conf` file in the installation etc directory with the following contents.

```
[DEFAULT]
greetingText = Howdy
```

Now lets run it.

```
jamest@calvin:~$ python helloworld -w bob
```

```
Howdy bob!  
Welcome
```

Lets tweak that ourapp.conf file a bit to add a more specific configuration section for this application.

```
[DEFAULT]  
greetingText = Howdy  
  
[helloworld]  
greetingText = Greetings
```

Now rerun the application

```
jamest@calvin:~$ python helloworld -w bob  
Greetings bob!  
Welcome
```

The format of the users \$HOME/.gnue/ourapp.conf and /usr/local/gnue/etc/ourapp.conf.fixed are the exact same as above. Try creating both these files and seeing how they interact.

A few things to note about the application above. More than one application can use the same CONFIGFILE setting. This allows families of applications to share a single config file. Each application can take advantage of the shared [DEFAULT] section yet have their own sections defined by the application="name" they set during initialization.

3. Using GNUe DataSources

Introduction

GNUe DataSources provide a mapping between a vendor-specific data storage method, most commonly a table, and a generic representation of that source of information. A GNUe DataSource hides the implementation of the storage of data from the developer. Mapping the generic manipulation of data to the appropriate vendor specific code needed to act upon the data. In the case of relational databases this would usually be SQL. But in dealing with Object Oriented data stores like GNUe-appserver it would be the appserver's custom protocol. The advantage of this approach is that an application developer you typically do not need to concern yourself with how the data is stored.

It's worth noting that while work is almost completed to allow the application programmer a method of using SQL in the creation of the datasource, it is not the preferred method of using them as it leads to the possibility of vendor lock-in.

Some of the key features of datasources include.

- Vendor neutral storage and retrieval of data. Backends can range from object servers, to relational databases, to flat files. GNUe-common hides the differences from the application programmer.
- Master/Detail relationships. This is the equivalent to linking tables in a relational database via primary/foreign key relationships. An application programmer can arbitrarily link multiple datasources to other datasources in complex relationships. Multiple detail datasources can link to a single master. Detail can also server as masters of other datasources. Navigation in the master datasource forces all details to keep in sync.
- Simple record caching. When loading huge datasets the datasource only loads records upon demand. Records are loaded in a quantity defined by the programmer to allow them to tailor the response time vs hits on the backend storage medium.
- XML representation. DataSources provide their own XML when using our XML-to-Object parser and Object-to-XML marshaller. Adding support for defining datasources along with conditionals in XML files requires only a few lines of code in your application.

Our First GDataSource Application

Lets start the introduction to gnue-common's GDataSources with a simple program.

The Task

We have a web server that takes registration information from contestants and stores it in a postgresSQL table. We want to transfer that registration information from the web server to our more secure internal server. Since a separate program will process the data, weed out duplicates, then move the data into the official list of registered applicants, we will not have to deal with those situations.

Before we begin with the application, we'll take a moment to make sure GNUe is setup properly and create some gnue-schema files that will generate the SQL we require.

Making Sure the Connection is Properly Setup

The first thing we should do is ensure that our connections are properly defined for use in GNUE. Assuming that GNUE has been installed in `/usr/local/gnue` we need to edit the file located at `/usr/local/gnue/etc/connections.conf`. We will add the following lines if they do not already exist.

```
[prod]
comment = Production DB
provider = psycopg
host = rdbms.example.com
dbname = prod
username = USERNAME
password = PASSWORD

[web]
comment = Web DB
provider = psycopg
host = www.example.com
dbname = web
username = USERNAME
password = PASSWORD
```

Lets break down the first entry `[prod]` in the file.

```
[prod]
```

This is the name of the GNUE connection. GNUE connection names are used throughout GNUE as a simple way to reuse connection information.

```
comment = Production DB
```

A more friendly text description of the connection. This is used in cases like GNUE-Designer's Wizards when it is prompting the user for the connection to use.

```
provider = psycopg
```

The provider name is a name assigned by GNUE to specify a vendor specific driver to use. In our case we are using the `psycopg` driver to access a postgresSQL database. See Appendix B for a list of valid drivers and corresponding GNUE provider names.

```
host = rdbms.example.com
```

The hostname or IP address of the machine hosting our postgresSQL database.

```
dbname = prod
```

The name of the postgresSQL database to use.

```
username = USERNAME
```

Username to use for database connection.

```
password = PASSWORD
```

Password to use for a database connection. Leaving "password =" for passwordless connection is possible.

It's worth noting that not all providers require the same settings. Appendix B lists the various supported databases and their specific connection.conf settings. For a more up to date listing visit <http://www.gnuenterprise.org/tools/common/databases>.

Defining the tables with gnu-schema

While gnu-schema will work quite well with pre-existing tables we will assume that we are going to need to create the registration table from scratch. Rather than generate the SQL to create the tables by hand I will instead use a gnu-common utility known as gnu-schema. gnu-schema allows you to take a table definition defined using an xml file and translate it to the vendor specific commands needed to generate the table. This process doesn't have to result in any specific format. For example, gnu-schema has a vendor target of HTML that produces an html page describing your table structure for documentation purposes.

Please Note: At the time of this writing gnu-schema is still a relatively new addition to gnu-common. As such it does not yet have output for all gnu-common supported vendor backends. You can get a list of supported backends by issuing the following command

```
jamest@calvin:~$ gnu-schema --help-connections
```

If your backend isn't supported then I would suggest picking one of the existing ones and editing the code to match your needs. Alternatively you could help us add support for your backend. :)

Let's look at the registration.gsd file we will be using to create our table structure.

```
<?xml version="1.0"?>
<schema>
  <tables>
    <!--
    *****
    Registration table
    *****
    -->
    <table name="registration">
      <fields>
        <field name="registration_id" type="key"
          defaultwith="serial" nullable="N"/>
        <field name="name" length="50" type="string" nullable="N" />
        <field name="address" type="string" length="50"/>
        <field name="address2" type="string" length="50"/>
        <field name="city" type="string" length="50"/>
        <field name="state" type="string" length="30"/>
        <field name="zip" type="string" length="20"/>
        <field name="zip4" type="string" length="4"/>
        <field name="phone_number" type="string" length="30"/>
      </fields>
      <primarykey name="pk_registration_id">
        <pkfield name="registration_id"/>
      </primarykey>
      <constraints/>
      <indexes/>
    </table>
  </tables>
</schema>
```

While we will cover the gsd format in greater detail in a later chapter we will take a moment to cover one of the less obvious lines.

```
<field name="registration_id" type="key"
  defaultwith="serial" nullable="N"/>
```

This line will cause gnue-schema to create a structure that autopopulates the registration_id field via a sequence if the backend will support it. The sequence will also be created automatically.

Now lets generate the vendor specific SQL we require. Since it is a relatively new utility in gnue-common, it doesn't have the same level of backend support as our datasource system.

```
jamest@calvin:~$ gnue-schema --connection web registration.gsd
Loading gsd file 'registration.gsd' ...
jamest@calvin:~$
jamest@calvin:~$ gnue-schema --connection prod registration.gsd
Loading gsd file 'registration.gsd' ...
jamest@calvin:~$
```

Since, as you see from the above transcript, we are using the same table structure on both the production database “prod” and the web server database “web”, we just need to use the vendor supplied SQL shell to run the generated SQL on both systems. Rather than step you through creating tables in your database I will just show the final result on the PostgreSQL system I'm currently using:

web=> \d registration

```

                                Table "public.registration"
  Column          |          Type          |          Modifiers
-----+-----+-----
 registration_id | bigint                 | not null default nextval
 ('registration_0_seq'::text)
 name            | character varying(50) | not null
 address        | character varying(50) |
 address2       | character varying(50) |
 city           | character varying(50) |
 state          | character varying(30) |
 zip            | character varying(20) |
 zip4           | character varying(4)  |
 phone_number   | character varying(30) |
Indexes:
    "pk_registration_id" primary key, btree (registration_id)

```

For completeness, here's also a complete MySQL example:

```
docelic@denali:~$ gnue-schema --createdb --connection web registration.gsd
Loading gsd file 'registration.gsd' ...
You are about to change the database 'web'. Continue [y,N]: y
Updating schema ...
docelic@denali:~$
docelic@denali:~$ gnue-schema --createdb --connection prod registration.gsd
Loading gsd file 'registration.gsd' ...
You are about to change the database 'prod'. Continue [y,N]: y
Updating schema ...
docelic@denali:~$
```

To be able to connect to the MySQL database, you can also adjust the user's ~/.my.cnf configuration file. Note that if the --createdb option is used, the same file must be tuned so

that running “mysqladmin create <dbname>” succeeds for the user, and that without further prompts. Here's a ~/.my.cnf example:

```
[client]
user = USERNAME
password = PASSWORD

[mysqladmin]
user = USERNAME
password = PASSWORD
```

And the necessary permissions for the user in MySQL database can be added this way:

```
docelic@denali:~$ sudo -H mysql -e 'grant all privileges on web.* to
  USERNAME@localhost identified by "PASSWORD"'
docelic@denali:~$ sudo -H mysql -e 'grant all privileges on prod.* to
  USERNAME@localhost identified by "PASSWORD"'
```

It is also possible to make gnue-schema only output the database specific SQL code, without actually making changes to the database itself. To achieve the effect, provide -f and -o OUTPUTFILE options to gnue-schema.

We are now going to need to put a sample record into the database “web”. While we could use gnue-designer to create a form to do this it is beyond the scope of this guide. Instead I'll show the SQL statement I used to populate my database.

```
web=> insert into registration (name, address, address2, city, state, zip, zip4, phone_number)
web-> values ('James Thompson','123 First',NULL,'Anytown','KS','11111','1111','785-555-1212');
INSERT 10969865 1
```

The rest of this page is blank so that the source code appears on a single page

Our Application

Now its time to take a look at the another application we'll be using to transfer the data. We will go ahead and use GClientApp as our base. The reasons will become apparent later in the chapter. Now a quick look at that code. The parts in bold will be the new features we've not covered in previous chapters and examples.

```

from gnue.common.apps.GClientApp import GClientApp
from gnue.common.datasources.GDataSource import DataSourceWrapper

class TransferTable(GClientApp):
    VERSION = "1.0.0"
    COMMAND = "tableTransfer"
    NAME = "tableTransfer"
    SUMMARY = ""
    Transfers data from one table in a datasource to
    a different table in the a datasource. The same
    datasource can be used in both tables.
    ""

    def run(self):
        SOURCE_DATASOURCE = 'web'
        SOURCE_TABLE = 'registration'
        DEST_DATASOURCE = 'prod'
        DEST_TABLE = 'registration'

        tableFields = ['registration_id', 'name', 'address', 'address2',
                       'city', 'state', 'zip', 'zip4', 'phone_number']

        sourceDatasource = DataSourceWrapper(self.connections,
                                             fields = tableFields,
                                             attributes={'connection': SOURCE_DATASOURCE,
                                                         'name': 'dtsSource',
                                                         'table': SOURCE_TABLE, } )

        destDatasource = DataSourceWrapper(self.connections,
                                           fields = tableFields,
                                           attributes={'connection': DEST_DATASOURCE,
                                                         'name': 'dtsDest',
                                                         'table': DEST_TABLE, } )

        # Perform the transfer
        sourceResultSet = sourceDatasource.createResultSet()
        destResultSet = destDatasource.createEmptyResultSet()

        for sourceRec in sourceResultSet:
            destRec = destResultSet.insertRecord()
            for field in tableFields:
                destRec[field] = sourceRec[field]
            sourceRec.delete()

        sourceResultSet.post()
        destResultSet.post()

        self.connections.commitAll()

if __name__ == '__main__':
    TransferTable().run()

```

Lets go through the code. We'll skip the code that isn't in bold. If you need to review what that code does please refer to chapter 2.

```

from gnue.common.datasources.GDataSource import DataSourceWrapper

```

DataSourceWrapper will allow us to properly access a GDataSource.

```

SOURCE_DATASOURCE = 'web'
SOURCE_TABLE = 'registration'
DEST_DATASOURCE = 'prod'
DEST_TABLE = 'registration'

tableFields = ['registration_id', 'name', 'address', 'address2',
               'city', 'state', 'zip', 'zip4', 'phone_number']

```

The uppercase variable names are simply used to make it easy to change values later. `tableFields` is a python list containing the field names we will be using from the tables.

```

sourceDatasource = DataSourceWrapper(self.connections,
                                     fields = tableFields,
                                     attributes={'connection': SOURCE_DATASOURCE,
                                               'name': 'dtsSource',
                                               'table': SOURCE_TABLE, } )

destDatasource   = DataSourceWrapper(self.connections,
                                     fields = tableFields,
                                     attributes={'connection': DEST_DATASOURCE,
                                               'name': 'dtsDest',
                                               'table': DEST_TABLE, } )

```

Here we are defining our two datasources. We'll only cover the `sourceDatasource` in detail.

The first argument passed in is **`self.connections`**, this is an instance of `gnue-common`'s connection manager. The connection manager does exactly as it's name implies. Since we're using `GClientApp` this is created for us at startup transparently.

Next we pass in the fields to the datasource **`fields = tableFields`**. This, in effect, registers the fields we are interested in manipulating with the `DataSource`. It's worth noting that if we were just reading data these names would not be needed. If a `fields` value is not passed in then the `GDataSource` will default to all fields in the table.

One piece of this code needs to be looked at in greater detail....

```

                                     attributes={'connection': DEST_DATASOURCE,
                                               'name': 'dtsDest',
                                               'table': DEST_TABLE, }

```

It maps values to attributes a `GDataSource` would normally have populated from it's XML definition. The value assigned to **`connection`** is the name of the `gnue` connection we previously defined in `/usr/local/gnue/etc/connections.conf`. **`name`** is the name `gnue` will internally assign to this datasource. It should be a unique name. While we do not use the defined name in this application the name would prove usefully if we were to use `gnue-commons` trigger system. The namespace provided by that system could contain this defined name as a handle to the datasource for your triggers. **`table`** is simply the name of the table to which the datasource will bind.

Next we move on to the code that will generate a pair `gnue-common`'s `ResultSets`.

```

sourceResultSet = sourceDatasource.createResultSet()
destResultSet   = destDatasource.createEmptyResultSet()

```

4. Using GParser

Introduction

GParser allows the application developer to define an DTD that maps XML files into the object instance tree in memory. At any time the tree can be converted back into its XML representation. When coupled with other common features such as triggers and datasources it will allow a developer to rapidly create file formats that can define or augment selected application's behavior.

Some of the key features of datasources include.

- The ability to define XML file formats via python dictionaries.
- Type checking of inputs
- XML file format validation. Single instance checking, parent/child relationships, required settings, default, etc are all configurable via the python dictionary.
- A flexible object tree phased initialization system via gnue-comon's GObj. Phases initialization allows an arbitrary number of passes through the object tree at at instance creation time.
- Several key features of gnue-common provide pre-defined GPaser settings making it trivial to add them to your definition.
- Auto-documentation. At this time it requires light alteration of gnue's doc-XML-attrs-openoffice.py script.

5. Using Triggers

Introduction

GNUe-common triggers provide a way of allowing the application programmer to define event hooks in their application that can run arbitrary python code defined external to the application. During execution, custom namespace is generated for triggers that provide additional features and functions to the trigger programmers without requiring them to learn the internals of your application. When utilized with gnue-common's GObj based objects and Gparser, the trigger namespace is easily populated with handles to all object instances in the GObj tree.

6. Using Events

Introduction

XX. A Complete GNUe-Common App

Appendix A: Schema Definition Elements

TODO

Schema Tags

schema

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
author	<i>text</i>		No DESCRIPTION PROVIDED
description	<i>text</i>		No DESCRIPTION PROVIDED
title	<i>text</i>		No DESCRIPTION PROVIDED
version	<i>text</i>		No DESCRIPTION PROVIDED

Child Nodes

data, tables

Data Tags

data

No DESCRIPTION PROVIDED

Child Nodes

tabledata

row

No DESCRIPTION PROVIDED

Child Nodes

value

rows

No DESCRIPTION PROVIDED

Child Nodes

row

tabledata

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		No DESCRIPTION PROVIDED
tablename	<i>text</i>		No DESCRIPTION PROVIDED

Child Nodes

rows

value

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
field	<i>text</i>		No DESCRIPTION PROVIDED
type	<i>text</i>	text	No DESCRIPTION PROVIDED

Tables Tags

tables

No DESCRIPTION PROVIDED

Child Nodes

import-table, table

constraint

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		No DESCRIPTION PROVIDED
type	<i>text</i>		No DESCRIPTION PROVIDED

Child Nodes

constraintfield, constraintref

constraintfield

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		No DESCRIPTION PROVIDED

constraintref

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		No DESCRIPTION PROVIDED
table	<i>text</i>		No DESCRIPTION PROVIDED

constraints

No DESCRIPTION PROVIDED

Child Nodes

constraint

field

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		No DESCRIPTION PROVIDED
type	<i>text</i>		No DESCRIPTION PROVIDED
auto	Y, N	N	No DESCRIPTION PROVIDED
default	<i>text</i>		No DESCRIPTION PROVIDED
defaultwith	constant, serial, timestam p	constant	No DESCRIPTION PROVIDED
description	<i>text</i>		No DESCRIPTION PROVIDED
length	<i>number</i>		No DESCRIPTION PROVIDED
nullable	Y, N	Y	No DESCRIPTION PROVIDED
precision	<i>number</i>	0	No DESCRIPTION PROVIDED

fields

No DESCRIPTION PROVIDED

Child Nodes

field, import-field

indexes

No DESCRIPTION PROVIDED

Child Nodes

index

indexfield

No DESCRIPTION PROVIDED

Attributes

<i>Attribute</i>	<i>Values</i>	<i>Default</i>	<i>Description</i>
name	text		No DESCRIPTION PROVIDED

pkfield

No DESCRIPTION PROVIDED

Attributes

<i>Attribute</i>	<i>Values</i>	<i>Default</i>	<i>Description</i>
name	text		No DESCRIPTION PROVIDED

primarykey

No DESCRIPTION PROVIDED

Attributes

<i>Attribute</i>	<i>Values</i>	<i>Default</i>	<i>Description</i>
name	text		No DESCRIPTION PROVIDED

Child Nodes

pkfield

table

No DESCRIPTION PROVIDED

Attributes

<i>Attribute</i>	<i>Values</i>	<i>Default</i>	<i>Description</i>
name	text		No DESCRIPTION PROVIDED
description	text		No DESCRIPTION PROVIDED

Child Nodes

constraints, fields, import-fields, indexes, primarykey

Import Tags

import-field

No DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	<i>text</i>		NO DESCRIPTION PROVIDED
name	<i>text</i>		NO DESCRIPTION PROVIDED
type	<i>text</i>		NO DESCRIPTION PROVIDED
auto	Y, N	N	NO DESCRIPTION PROVIDED
default	<i>text</i>		NO DESCRIPTION PROVIDED
defaultwith	constant, serial, timestamp	constant	NO DESCRIPTION PROVIDED
description	<i>text</i>		NO DESCRIPTION PROVIDED
length	<i>number</i>		NO DESCRIPTION PROVIDED
nullable	Y, N	Y	NO DESCRIPTION PROVIDED
precision	<i>number</i>	0	NO DESCRIPTION PROVIDED

import-fields

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	<i>text</i>		NO DESCRIPTION PROVIDED

import-table

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	<i>text</i>		NO DESCRIPTION PROVIDED
name	<i>text</i>		NO DESCRIPTION PROVIDED
description	<i>text</i>		NO DESCRIPTION PROVIDED

index

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	<i>text</i>		NO DESCRIPTION PROVIDED
unique	Y, N	N	NO DESCRIPTION PROVIDED

Child Nodes

indexfield

Appendix B: Datasource Drivers

Datasources can support various types of backends. Currently GNUe Common supports datasources based upon popular databases and some middleware.

Supported Databases

PostgreSQL

PyGreSQL [<http://druid.net/pygresql/>]

POSIX Support	YES
Win32 Support	YES [no prebuilt binaries]
In Prebuilt Packages	No
Platforms Tested	GNU/Linux [Debian 2.x, Slackware 8.0, RedHat 6.x/7.x]

Description:

PyGreSQL is the standard PostgreSQL-Python driver. Because PyGreSQL predates Python's DB-API 2.0 specification, its support of that standard is hap-hazard at best -- you might want to consider one of the other drivers below as they were written with this support in mind.

Example connections.conf entry:

```
[myconn]
```

```
provider=postgresql # Use the pygresql adapter
```

```
host=localhost # The hostname/IP of the postgresql host
```

```
dbname=mydb # The name of the pg database
```

```
encoding=ASCII # Optional: the client encoding for PG
```

PyPgSQL [<http://pypgsql.sf.net/>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries]
In Prebuilt Packages	NO

Platforms Tested
GNU/Linux [Debian 2.x, Slackware 8.0, RedHat 6.x/7.x]
Windows 98

Description:

Written by Billy Allie, PyPgSQL is a database interface for PostgreSQL 7.x.

Example connections.conf entry:

[myconn]

provider=pypgsql # Use the pypgsql adapter

host=localhost # The hostname/IP of the postgresql host

dbname=mydb # The name of the pg database

encoding=ASCII # Optional: the client encoding for PG

psycopg [<http://initd.org/Software/psycopg/>]

POSIX Support	YES
Win32 Support	??? (We need your help!)
In Prebuilt Packages	NO
Platforms Tested	??? (We need your help!)

Description:

From the Psycopg website: "It was written from scratch with the aim of being very small and fast, and stable as a rock." Written by initd.org volunteers.

Example connections.conf entry:

[myconn]

provider=psycopg # Use the psycopg adapter

host=localhost # The hostname/IP of the postgresql host

dbname=mydb # The name of the pg database

encoding=ASCII # Optional: the client encoding for PG Notes:

1. Available in Debian [sid] as: "apt-get install python2-psycopg"
2. This driver has not been fully tested. If you are successfully using this driver, please let us know.

PoPy [<http://popy.sf.net/>]

POSIX Support	YES
Win32 Support	??? (We need your help!)
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by Thierry Michel and friends. Strong multi-thread support.

Example connections.conf entry:

[myconn]

provider=popy # Use the PoPy adapter

host=localhost # The hostname/IP of the postgresql host

dbname=mydb # The name of the pg database

encoding=ASCII # Optional: the client encoding for PG Notes:

1. Available in Debian [sid] as: "apt-get install python-popy"
2. This driver has not been fully tested. If you are successfully using this driver, please let us know.

MySQL

MySQLdb [<http://sourceforge.net/projects/mysql-python/>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries at http://www.cs.fhm.edu/~ifw00065/]

In Prebuilt Packages	
Platforms Tested	GNU/Linux [RedHat 7.x]

Description:

Written by Andy Dustman, this driver supports MySQL 3.22, 3.23, and 4.x.

Example connections.conf entry:

[myconn]

provider=mysql # Use the MySQLdb adapter

host=localhost # The hostname/IP of the MySQL host

dbname=mydb # The name of the MySQL database Notes:

1. Transactions are supported if MySQL is compiled with proper transactional support (3.x series does NOT do this by default!)
2. This driver has been partially tested. If you are successfully using this driver, please let us know.
3. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

Oracle

DCOracle [<http://www.zope.org/Members/matt/dco2>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries (WinNT only)]
In Prebuilt Packages	No - License Issues OCI Library
Platforms Tested	GNU/Linux [RedHat 7.x]

Description:

An Oracle driver from Digital Creations (Zope). Works with Oracle 7.3 and 8.x via Oracle's SQL-Net OCI interface.

Example connections.conf entry:

[myconn]

provider=oracle # Use the DCOracle2 adapter

service=mytnsname # The TNS connection string of the database Notes:

1. Requires Oracle Client Libraries
2. Does not recognize the TWO_TASK environment setting.
3. This driver has been partially tested. If you are successfully using this driver, please let us know.

cxOracle [<http://www.computronix.com/utilities/>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries
In Prebuilt Packages	No - License Issues OCI Library
Platforms Tested	??? (We need your help!)

Description:

An Oracle driver from Computronix. Works with Oracle 7.3 and 8.x via Oracle's SQL-Net OCI interface.

Example connections.conf entry:

[myconn]

provider=cxoracle # Use the CX Oracle adapter

service=mytnsname # The TNS connection string of the database Notes:

1. Requires Oracle Client Libraries.
2. Does not recognize the TWO_TASK environment setting.
3. This driver has not been fully tested. If you are successfully using this driver, please let us know.

IBM DB/2

DB2-Python [<ftp://people.linuxkorea.co.kr/pub/DB2/>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries]
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

A DB/2 driver from Bryan Lee. Works with IBM's DB/2 version 7.1.

Example connections.conf entry:

[myconn]

provider=db2 # Use the DB/2 adapter

service=mydsn # The DSN connection string of the database Notes:

1. The Win32 binaries are included in the tar'ed file (in win32/)
2. This driver has not been fully tested. If you are successfully using this driver, please let us know.
3. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

SAP-DB

SAP-DB [<http://www.sapdb.org/>]

POSIX Support	YES
Win32 Support	???
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by the SAP-DB people, this driver is part of the SAP DB distribution. Works with the 7.x series of SAP-DB

Example connections.conf entry:

[myconn]

provider=sapdb # Use the SAP-DB adapter

host=localhost # The hostname/IP of the SAP-DB host

dbname=mydb # The name of the SAP-DB database Notes:

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

Interbase

Kinterbasdb [<http://kinterbasdb.sf.net/>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries]
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by Alexander Kuznetsov, this driver provides support for Interbase.

Example connections.conf entry:

[myconn]

provider=informix # Use the Kinfxdb adapter

host=localhost # The hostname/IP of the Informix host

dbname=mydb # The name of the Informix database

Notes:

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.

2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

OpenIngres, CA Ingres, CA Ingres II

ingmod [<http://www.informatik.uni-rostock.de/~hme/software/>]

POSIX Support	YES
Win32 Support	???
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by Holger Meyer, this driver supports OpenIngres, as well as the various CA Ingres solutions.

Example connections.conf entry:

```
[myconn]
```

```
provider=ingres # Use the ingmod adapter
```

```
dbname=mydb # The name of the Ingres database Notes:
```

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

Informix

Kinfxdb [<http://thor.prohosting.com/~alexan/pub/Kinfxdb/Kinfxdb-0.2.tar.gz>]

POSIX Support	YES
Win32 Support	???
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by Alexander Kuznetsov, this driver provides support for Informix.

Example connections.conf entry:

[myconn]

provider=informix # Use the Kinfxdb adapter

host=localhost # The hostname/IP of the Informix host

dbname=mydb # The name of the Informix database Notes:

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

Sybase

Sybase [<http://www.object-craft.com.au/projects/sybase/>]

POSIX Support	YES
Win32 Support	YES [no prebuilt binaries]
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

Written by David Cole, this driver works with the 11.x.x series of Sybase ASE.

Example connections.conf entry:

[myconn]

provider=sapdb # Use the SAP-DB adapter

host=localhost # The hostname/IP of the SAP-DB host

dbname=mydb # The name of the SAP-DB database Notes:

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

3.

Detailed Middleware Listings

GNUe Enterprise App Server

GEAS [<http://www.gnue.org/>]

POSIX Support	YES
Win32 Support	NO
In Prebuilt Packages	
Platforms Tested	

Description:

GNUe's own Enterprise Application Server, currently under heavy development, provides a business rules and data backend. Works with a variety of databases. [Not yet stable]

Example connections.conf entry:

```
[myconn]
```

```
provider=geas # Use the GEAS adapter
```

```
host=localhost # The hostname/IP of the GEAS host
```

```
dbname=mydb # The name of the GEAS database Notes:
```

1. Non-functional in the 0.1.0 release of gnue-common
2. Currently only recommended for developer's use.
3. GNUe-Common connects to GEAS via ORBit, so you will need an ORBit installation on each client.

ODBC

PythonWin ODBC

[<http://aspn.activestate.com/ASPN/Downloads/ActivePython/Extensions/Win32all>]

POSIX Support	NO
Win32 Support	YES [prebuilt binaries]
In Prebuilt Packages	YES [Win32 only]
Platforms Tested	Windows 98

Description:

A public domain ODBC interface for Python. Supports practically any database backend with Windows

ODBC drivers.

Example connections.conf entry:

```
[myconn]
```

```
provider=odbc # Use the ODBC adapter
```

```
service=mydsn # The DSN connection string of the database Notes:
```

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not yet support schema introspection, so you will not be able to use Wizards in GNUe Designer.

mxODBC [<http://www.lemburg.com/files/python/mxODBC.html>]

POSIX Support	YES
Win32 Support	YES [prebuilt binaries]
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

A commercial ODBC interface for Python written by Marc-André Lemburg. Supports practically any database backend with ODBC drivers (are there any without ODBC drivers?)

Example connections.conf entry:

```
[myconn]
```

```
provider=mxodbc # Use the mxODBC adapter
```

```
service=Subdriver|mydsn # The DSN connection string of the database Notes:
```

1. The mxODBC driver is only free for non-commercial use! Check their license before using.
2. Uses Windows ODBC drivers natively, but on Unix, requires subpackages to be installed. Under Windows, the Subdriver (in the service=entry) will always be "Windows" (e.g., "service=Windows|mydsn") However, under Unix, the Subdriver will be the subpackage name (e.g, "PostgreSQL", "Oracle", "iODBC", "unixODBC") See the mxODBC documentation for more information.
3. This driver has not been fully tested. If you are successfully using this driver, please let us know.
4. This driver does not support schema introspection, so you will not be able to use Wizards in GNUe Designer.

SQLRelay

SQLRelay [<http://www.firstworks.com/sqlrelay.html>]

POSIX Support	YES
Win32 Support	???
In Prebuilt Packages	
Platforms Tested	??? (We need your help!)

Description:

From FirstWork's website: "SQL Relay is a persistent database connection pooling, proxying and load balancing system for Unix and Linux."

Example connections.conf entry:

```
[myconn]
```

```
provider=sqlrelay # Use the DB/2 Oracle adapter
```

```
host=hostname:port # The host and port running SQLRelay Notes:
```

1. This driver has not been fully tested. If you are successfully using this driver, please let us know.
2. This driver does not support schema introspection, so you will not be able to use Wizards in GNUe Designer.

Command Line Options

GNUe Form's `gnue-forms` accepts a variety of command line options. These options can always be viewed by typing:

```
gnue-forms --help
```

at a command prompt. Not all options work on all platforms (e.g., disabling the splash screen has no effect on a text based interface.)

7.3.2.

<code>--connections <loc></code>	Specifies an alternate location of a connection definition file. <code><loc></code> may specify a file name (<code>/usr/local/gnue/etc/connections.conf</code>), or a URL location (<code>http://localhost/connections.conf</code>). If this option is not specified, the environment variable <code>GNUE_CONNECTIONS</code> is checked.
--	--

<code>--debug-file <file></code>	Sends all debugging messages to a specified file (e.g., " <code>--debug-file trace.log</code> " sends all output to "trace.log")
<code>--debug-level <level>, -d</code>	Enables debugging messages. Argument specifies the level of messages to display (e.g., "-d5" displays all debugging messages at level 5 or below.)
<code>--help</code>	Displays a help screen.
<code>--no_splash, -s</code>	Disables the splash screen.
<code>--profile</code>	Run Python's built-in profiler and display the resulting run statistics.
<code>--user_interface <type>, -u</code>	The currently supported values for <type> are <code>gui</code> , <code>text</code> (unstable), and <code>pytext</code>
<code>--version</code>	Displays the current version information for this program.

Examples:

1. Run the form "samples.gfd" without the opening splash screen:

```
gnue-forms --no_splash samples.gfd
```

2. Run the form "test.gfd" located on the company's web server in debugging level 10:

```
gnue-forms -d 10 http://my.company.com/forms/samples.gf
```


Alphabetical Index