

CREACIÓN DE UNA APLICACIÓN WEB (FORMULARIO)

Descripción de la aplicación que se va a construir

Vamos a construir una aplicación en varias partes. El objetivo consiste en mostrar cómo una aplicación puede tener una interfaz web, una interfaz gráfica y una interfaz de consola y, además, compartir los datos.

La aplicación, en su globalidad, es un simple formulario de contacto.

La solución web que vamos a utilizar es Pyramid, un framework web que utiliza distintos módulos de Python bien contrastados y sobre los que se basa para ofrecer una solución fiable, pero flexible, pues en todo momento es posible remplazar alguno de los módulos por otro (por ejemplo, cambiar el módulo que gestiona el templating, o el que gestiona la autenticación, o incluso el que gestiona el enrutado...). El desacople de estos módulos es una de las ventajas de este framework.

La solución para gestionar la persistencia que vamos a utilizar es PostgreSQL.

Cabe destacar que se puede utilizar también TurboGears. Esta solución sigue siendo excelente e implementa varios componentes similares a Pyramid así como muchos otros bastante específicos e interesantes, pero se parece más a un framework como Django.

Este último es un framework full-stack, es decir, que integra en su seno todos los componentes elementales (ya no es necesario utilizar proyectos complementarios para realizar las tareas básicas y, por otro lado, el desarrollar ya no tiene restricciones).

La gran ventaja de Django es que dispone de una galaxia impresionante de módulos externos, que permiten realizar tareas complejas con muy poco esfuerzo. Django es uno de los productos de referencia de la Web y contribuye en gran medida a la popularidad de Python.

Destacaremos que es posible utilizar el ORM SQLAlchemy con Django, ya que se ha presentado en este libro, y que también es posible utilizar otro sistema de plantillas diferente al que se provee de base.

A contrario que Django, Pyramid es un framework minimalista. Para construir una aplicación es necesario agregar varios componentes Python independientes. La ventaja es que cada uno es intercambiable. Se trata de un framework muy ligero llamado light-stack.

En efecto, el desarrollador de Python puede elegir cada componente y debe realizar la integración. El especialista de Python tenderá a dar prioridad a un framework determinado, que le permitirá reutilizar componentes que ya utilice en algún otro proyecto y construir su solución personalizada y reutilizable.

Por el contrario, los desarrolladores que debuten tendrán que hacer un mayor esfuerzo para encontrar los componentes adecuados (los foros y la experiencia de la comunidad ayudan mucho), aunque quien sea curioso se formará muy rápidamente.

La aplicación que vamos a diseñar tiene como objetivo mostrar un formulario de contacto y registrar los datos en una base de datos relacional.

Esta misma aplicación se desarrollará para permitir reutilizar el modelo de datos para la creación de una interfaz gráfica, que será el tema central del capítulo

Crear una aplicación gráfica en 20 minutos, así como la creación de una aplicación terminal

Crear una aplicación de consola en 10 minutos.

Las tres aplicaciones utilizarán la misma base de datos.

Los 30 minutos se entienden como el tiempo de desarrollo, una vez realizada la fase de diseño, pensada la estructura de la base de datos, así como las relaciones entre objetos e interfaces.

Además de la implementación de Pyramid, este tiempo se dedicará para realizar:

- el modelo de objetos: 5 objetos, es decir 5 clases,
- un formulario,
- la modificación del controlador del índice y de la plantilla asociada,
- la creación de cuatro controladores,
- la creación de dos plantillas.

Implementación

1. Aislar el entorno

Cuando se crea un proyecto, conviene crear una carpeta que contendrá Python (si es preciso con una versión concreta, correspondiente a la versión de producción o de la futura producción), así como todas sus dependencias.

De este modo, se minimiza la probabilidad de sufrir una anomalía en la producción que no sea reproducible en un entorno local.

He aquí cómo crear dicho entorno:

```
$ virtualenv -p python3 nombre_entorno
```

```
$ cd nombre_entorno/
```

En esta carpeta se desarrollará el proyecto.

A continuación, es necesario habilitar este entorno. En Linux:

```
$ source bin/activate
```

En Windows:

```
Scripts\activate.bat
```

Una vez activado el entorno, la instalación de un módulo de Python agrega este módulo al Python local, no al Python de la máquina. Y de manera recíproca, algunos módulos instalados en la máquina pueden no estar disponibles y habría que reinstalarlos (con **pip_install**) o puede que no estén en la versión correcta (habría que actualizarlos con **pip_install -U**).

Cuando se cierra el terminal, el entorno se desactiva; en caso contrario, la creación de este entorno agregaría también el comando **deactivate**, que basta para volver a un terminal normal.

2. Creación del proyecto

La primera etapa consiste en instalar Pyramid, con **easy_install**, una vez dentro del entorno virtual creado anteriormente:

```
$ pip install pyramid
```

Si funciona correctamente, el siguiente comando debería funcionar:

```
$ pserve -help
```

También es posible verificar de dónde viene el comando utilizado (en caso de que exista también en el sistema):

```
$ which pserve
```

```
/path/to/nombre_entorno/bin/pserve
```

Es posible, también, realizar la misma verificación para **pip** o cualquier otro comando que utilicemos más adelante, pero especialmente para el propio comando **Python**. Además, si el programa se llama **Python**, se trata efectivamente de **Python 3**.

```
$ python -version
```

Python 3.2.3

pserve es el servidor web Python que permite ejecutar la aplicación con el objetivo de desarrollarla, ofreciendo herramientas para el desarrollo (no sirve para poner la aplicación en producción, pues ese es el rol de apache2+wsgi).

Tan solo queda crear el proyecto:

```
$ pcreate -s alchemy contact
```

3. Configuración

Una vez creado el proyecto, hay que ubicarse en la carpeta que contiene nuestro proyecto:

```
$ cd contact/
```

La primera etapa consiste en abrir el archivo **setup.py** y personalizarlo:

```
$ vi setup.py
```

He aquí los campos que es preciso modificar, entre otros:

- version
- description
- author
- author_email
- install_requires (agregar psycpg2)
- paster_plugins
- tests_require
- message extractors
- entry_points

Los últimos puntos deben modificarse cuando se conocen bien los componentes y se quieren aportar los cambios.

A continuación, hay que crear la base de datos; hemos seleccionado postgres:

- utilizar pgAdminIII.
- crear el usuario «py» con la contraseña «pypass», asignándole el permiso de usuario sencillo.
- crear la base de datos llamada «contact» declarando a «py» como propietario.

A continuación, hay que configurar la aplicación conforme a los siguientes datos:

```
$ vim development.ini
```

Los campos que debemos modificar son:

- sqlalchemy.url escribiendo: «postgres://py:pypass@localhost:5432/contact
- sqlalchemy.*, a nuestra conveniencia
- debug, host, port

A continuación, construiremos la aplicación a partir de estas declaraciones:

```
$ python setup.py develop
```

Y crearemos la base de datos a partir de los modelos existentes por defecto:

```
$ initialize_contact_db development.ini
```

El servidor (que todavía no hace nada) puede ejecutarse:

```
$ pserve development.ini
```

4. Primeros ensayos

Ahora es necesario conectarse a la aplicación con el navegador web preferido escribiendo la URL correspondiente a la configuración (por defecto, localhost:6543).

Es posible ver la página de inicio, creada automáticamente.

A continuación, es necesario leer los archivos que se encuentran en la carpeta `contact` para ver los modelos, controladores y templates con objeto de comprender sus roles y lo que hacen.

A nivel de las vistas, es necesario aprender a controlar los lenguajes de template utilizados. Para comenzar, los templates son XHTML stricto y no respetar la norma supone un error (olvidar una etiqueta de cierre, por ejemplo, o utilizar un nombre de etiqueta deprecado...). El lenguaje por defecto es genshi (<http://genshi.edgewall.org/>) y requiere aprender ciertas nociones que son fáciles de dominar, puesto que son coherentes con el lenguaje Python.

Para los modelos, se utiliza simplemente SQLAlchemy, que hemos visto en la sección dedicada a SQL. Basta con saber que la creación del objeto de sesión y la del objeto metadata ya están realizadas y basta con agregar los propios objetos.

Por último, en lo relativo a los controladores, se trata de funciones que reciben como parámetro un único parámetro, que es la consulta, y que devuelven un diccionario o terminan con una redirección.

Mediante un decorador, estas funciones se agregan a una ruta (dicho de otro modo, a una declaración de URL) y a un template. Es posible, también, utilizar estos controladores como servicios web.

Realizar la aplicación

A continuación, pasamos a realizar la aplicación. Para ello, vamos a empezar presentando los modelos en su integridad y, a continuación, las vistas para mostrar cómo se organizan y muestran los datos; por último veremos los controladores para manipularlos y vincular las vistas al modelo.

1. Modelos

Esta sección es particular, pues este modelo sirve también como base a los proyectos detallados en los siguientes capítulos. Vamos a escribir el modelo en un archivo «model.py» situado en la carpeta contact.

Contiene:

```
# -*- coding: utf-8 -*-
```

```
from datetime import datetime
```

```
from sqlalchemy import (
```

```
    Column,
```

```
    ForeignKey,
```

```
    Index,
```

```
    Integer,
```

```
    UnicodeText,
```

```
    Unicode,
```

```
    DateTime,
```

```
)
```

```
from sqlalchemy.ext.declarative import declarative_base
```

```
from sqlalchemy.orm import (
```

```
    scoped_session,
```

```
    sessionmaker,
```

```
)
```

```
from zope.sqlalchemy import ZopeTransactionExtension
```

```
DBSession = scoped_session(sessionmaker  
(extension=ZopeTransactionExtension()))
```

```
Base = declarative_base()
```

```
__all__ = ['Subject', 'Contact']
```

```
class Subject(Base):
```

```
    __tablename__ = 'subjects'
```

```
    id = Column(Integer, primary_key=True)
```

```
    name = Column(Unicode(255), unique=True)
```

```
Index('subject_name_index', Subject.name, unique=True,  
mysql_length=255)
```

```
class Contact(Base):
```

```
    __tablename__ = 'contacts'
```

```
    id = Column(Integer, primary_key=True)
```

```
    email = Column(Unicode(255), nullable=False)
```

```
    subject_id = Column(Integer, ForeignKey(Subject.id))
```

```
    text = Column(UnicodeText, nullable=False)
```

```
    created = Column(DateTime, default=datetime.now)
```

```
Index('contact_email_index', Contact.email, mysql_length=255)
```

Es necesario conocer SQLAlchemy es un requisito previo, el cual se explica en el capítulo Manipulación de datos de este libro.

Este ejemplo muestra cómo hacer que un campo sea obligatorio o único, cómo colocar índices, claves primarias y claves foráneas o valores por defecto, como la fecha en curso.

Una vez escritos los modelos, se modifica el script de inicialización de datos que se encuentra en **contact/scripts/initializedb.py** de la siguiente manera:

```
import os

import sys

import transaction

from sqlalchemy import engine_from_config

from pyramid.paster import (
    get_appsettings,
    setup_logging,
)

from pyramid.scripts.common import parse_vars

from ..models import (
    DBSession,
    Subject,
    Contact,
    Base,
)

def usage(argv):
    cmd = os.path.basename(argv[0])
```



```
print('usage: %s <config_uri> [var=value]\n'  
      '(example: "%s development.ini")' % (cmd, cmd))  
  
sys.exit(1)
```

```
def main(argv=sys.argv):  
    if len(argv) < 2:  
        usage(argv)  
  
    config_uri = argv[1]  
  
    options = parse_vars(argv[2:])  
  
    setup_logging(config_uri)  
  
    settings = get_appsettings(config_uri, options=options)  
  
    engine = engine_from_config(settings, 'sqlalchemy.')  
  
    DBSession.configure(bind=engine)  
  
    Base.metadata.drop_all(engine)  
  
    Base.metadata.create_all(engine)    with transaction.manager:  
  
    for subject_name in ('Python', 'Pyramid', 'Pygame'):  
  
        model = Subject(name=subject_name)  
  
        DBSession.add(model)  
  
    model = Contact(email='me@example.com', text='Why 42 is  
  
the answer ?')  
  
    DBSession.add(model)
```

A continuación hay que crear la base de datos con nuevas tablas. No obstante, el modelo MyModel ya no existe, de modo que la instrucción **drop_all** no funcionará sobre este modelo, y habrá que eliminarlo manualmente.

Para volver a crear esta base de datos, es necesario detener el servidor y hacerlo exactamente de la misma manera que antes:

```
$ initialize_contact_db development.ini
```

Una vez realizado, se reinicia el servidor:

```
$ pserve --reload development.ini
```

La opción **--reload** inicia un monitor que supervisa los archivos del proyecto y reinicia el servidor cada vez que se modifica alguno de ellos, lo cual resulta muy práctico durante la fase de desarrollo.

A continuación mostraremos un formulario vacío que permitirá introducir datos de manera amigable. El resto del tiempo se utilizará para modificar la página de índice y crear los controladores.

2. Vistas

El modelo es bastante independiente, el único cambio consiste en manipular los datos y sus relaciones con total libertad, las vistas y el controlador son relativamente interdependientes en el sentido de que el controlador está al servicio de la vista y debe proveer los datos que espera.

A continuación presentaremos la lógica de visualización que dirige las selecciones realizadas para mostrar esta vista y solo a continuación podremos ver los controladores; es útil ver uno para comprenderlo y conocer las interacciones entre ambos.

Cuando se arranca un nuevo proyecto con Pyramid, se crea por defecto una página de índice que es rosada, basada en un framework CSS que es Twitter Bootstrap, uno de los frameworks más utilizados y populares a día de hoy.

Si desea utilizar otro framework, siéntase libre de integrarlo: bastará con modificar totalmente la estructura de la página, cambiando simplemente los archivos CSS y JavaScript referenciados.

En lo relativo a nuestro proyecto, vamos a utilizar las posibilidades ofrecidas por este framework y modificar únicamente la parte de contenido. La primera actividad consiste en modificar la frase de bienvenida:

```
<div class="content">

    <h1><span class="font-semi-bold">Pyramid - Contact

Ejemplo</span> <span class="smaller">starter template</span></h1>

    <p class="lead">Welcome to

<span class="font-normal">${project}</span>,

an&nbsp;application generated&nbsp;by<br>the

<span class="font-normal">Pyramid Web Framework</span>.</p>
```

A continuación guardaremos algo de sitio para alojar un mensaje informativo (que permite comunicar que el formulario se ha procesado):

```
<div class="alert alert-info" tal:condition="infos"
```

```
tal:repeat="info infos" tal:content="info">Information messages</div>
```

En este punto, es necesario realizar una pequeña explicación. Por defecto, Pyramid utiliza un motor de visualización llamado Chameleon (camaleón). Se trata de un lenguaje de template heredado de Zope que permite visualizar datos de manera dinámica. En nuestro ejemplo, se muestran tantos párrafos de información como información deba mostrarse. Los atributos están prefijados por tal. Se utiliza, de hecho, los lenguajes METAL/TAL/TALES.

Este lenguaje de templates es muy potente y permite cubrir todas las necesidades más habituales. Es, también, posible utilizar otros lenguajes tales como Genshi o Jinja, por ejemplo.

A continuación, se mostrará el formulario utilizando una estructura adaptada a Twitter Bootstrap, es decir reproduciendo los ejemplos provistos por su sitio de referencia: <http://getbootstrap.com/css/> y utilizando las clases adecuadas:

```
<form role="form" class="form-horizontal" method="POST">
```

```
<div class="form-group">
```

```
<label for="email" class="col-sm-2 control-label">
```

```
Your e-mail</label>
```

```
<div class="col-sm-10">
```

```
<input type="email" name="email" class="form-control" />
```

```
</div>
```

```
</div>
```

```
<div class="form-group">
```

```
<label for="subject" class="col-sm-2 control-label">
```

```
Pick a subject</label>
```

```
<div class="col-sm-10">
```

```
<select name="subject_id" class="form-control">
```

```
<option tal:repeat="subject subjects"
```

```
tal:attributes="value subject.id" tal:content="subject.name"></option>
```

```
</select>
```

```
</div>
```

```
</div>

<div class="form-group">

  <label for="text" class="col-sm-2 control-label">Your
message</label>

  <div class="col-sm-10">

    <textarea name="text" class="form-control"></textarea>

  </div>

</div>

<div class=" col-sm-offset-2 col-sm-10">

  <button type="submit" class="btn btn-default">

Submit</button>

  </div>

</form>

</div>
```

Destaca también la iteración sobre las opciones de la lista de selección que permite rellenar dinámicamente los asuntos disponibles.

3. Controladores

Un controlador manipula el ORM para acceder a los datos, para transmitirlos a la vista, y también para recuperar la información ofrecida por el usuario de la aplicación, dotarla de seguridad y almacenarla de manera persistente.

Para ello, el controlador utiliza la sesión vinculada al ORM; conviene leer el capítulo Bases de datos, sección SQLAlchemy para saber cómo utilizarla correctamente. Antes de explicar el funcionamiento de los controladores, es útil interesarse por el proceso bootstrap, es decir, la manera en la que se construye la aplicación y se articula para generar su propio entorno e invocar al controlador adaptado.

Vamos a reproducir aquí todas las etapas para comprender lo que ocurre, dado que será útil para el siguiente capítulo.

La primera etapa consiste en inicializar un entorno que se cree automáticamente en Pyramid:

```
>>> from pyramid.paster import bootstrap
```

```
>>> env = bootstrap('development.ini')
```

```
>>> settings, closer = env['registry'].settings, env['closer']
```

Como puede observarse, se va a buscar dos elementos, los parámetros, así como un elemento que nos va a permitir cerrar apropiadamente nuestro entorno cuando se hayan terminado de realizar las manipulaciones. A continuación, podemos configurar el motor SQLAlchemy a partir de la información provista en el archivo de configuración:

```
>>> from sqlalchemy import engine_from_config
```

```
>>> engine = engine_from_config(settings, 'sqlalchemy.')
```

Tan solo queda configurar la sesión y la base SQLAlchemy:

```
>>> from contact.models import DBSession, Base, Contact
```

```
>>> DBSession.configure(bind=engine)
```

```
>>> Base.metadata.bind = engine
```

Probamos (configuración con **sqlalchemy.echo** a **True**):

```
>>> DBSession.query(Contact).all()
```

```
2014-03-04 23:42:25,864 INFO sqlalchemy.engine.base.Engine select version()
```

```
2014-03-04 23:42:25,875 INFO sqlalchemy.engine.base.Engine {}
```

```
2014-03-04 23:42:25,876 INFO sqlalchemy.engine.base.Engine  
select current_schema()
```

```
2014-03-04 23:42:25,876 INFO sqlalchemy.engine.base.Engine {}
```

```
2014-03-04 23:42:25,877 INFO sqlalchemy.engine.base.Engine SELECT  
CAST('test plain returns' AS VARCHAR(60)) AS anon_1
```

```
2014-03-04 23:42:25,877 INFO sqlalchemy.engine.base.Engine {}
```

```
2014-03-04 23:42:25,878 INFO sqlalchemy.engine.base.Engine SELECT  
CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
```

```
2014-03-04 23:42:25,878 INFO sqlalchemy.engine.base.Engine {}
```

```
2014-03-04 23:42:25,878 INFO sqlalchemy.engine.base.Engine show  
standard_conforming_strings
```

```
2014-03-04 23:42:25,878 INFO sqlalchemy.engine.base.Engine {}
```

```
2014-03-04 23:42:25,879 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
```

```
2014-03-04 23:42:25,879 INFO sqlalchemy.engine.base.Engine SELECT
```

```
contacts.id AS contacts_id, contacts.email AS contacts_email,
```

```
contacts.subject_id AS contacts_subject_id, contacts.text AS
```

```
contacts_text, contacts.created AS contacts_created FROM contacts
```

```
2014-03-04 23:42:25,879 INFO sqlalchemy.engine.base.Engine {}
```

```
[<contact.models.Contact object at 0x5467f50>,
```

```
<contact.models.Contact
```

```
object at 0x546af10>, <contact.models.Contact object at 0x546a290>,
```

```
<contact.models.Contact object at 0x546a110>,
```

```
<contact.models.Contact
```

```
object at 0x546a050>, <contact.models.Contact object at 0x546af90>]
```

Obtenemos un entorno de trabajo y de pruebas correcto.

No debemos olvidar salir del entorno convenientemente:

```
>>> closer()
```

Una vez que sabemos manipular la base de datos, basta con crear los controladores. Se trata de simples funciones que están decoradas para precisar la ruta a la que corresponden, así como el método, y el hecho de que sea Ajax o no, así como el nombre del método o de los métodos aceptados.

De este modo, una misma ruta puede utilizarse de diversas maneras y corresponder a diversas acciones. Es posible tener, por ejemplo, una ruta que permita realizar una autenticación que funcione con Ajax o según la web clásica. En lo relativo a los métodos HTTP utilizados por los navegadores, el método GET permite mostrar el formulario y el método POST permite procesarlo (solo los formularios de búsqueda se procesan también con GET, cuyos resultados pueden alojarse en caché).

Para los servicios web, se distingue el método GET, que permite obtener información, el método POST, que permite agregar información, el método PUT, que permite modificar información (la distinción es importante), el método DELETE, que permite eliminar algún dato y, por último, el método HEAD, que permite recuperar únicamente los encabezados (es imposible colocar un contenido).

Volviendo a los controladores, cuando tienen como objetivo devolver un resultado, como el hecho de mostrar una página, se devuelve un único diccionario como resultado del método. El decorador se encarga de ubicar los datos en la vista para generarla (o, en los servicios web, de transformar este diccionario en Ajax). He aquí los métodos que se propone escribir para mostrar el formulario y procesarlo:

```
from pyramid.response import Response
```

```
from pyramid.httpexceptions import HTTPFound

from pyramid.view import view_config


import transaction


from .models import (

    DBSession,

    Subject,

    Contact,

)


@view_config(route_name='home', request_method='GET',
renderer='templates/mytemplate.pt')

def contact_get(request):

    """Display the contact form"""

    subjects = DBSession.query(Subject).all()

    infos = request.session.pop_flash('infos')

    return {'subjects': subjects, 'project': 'contact', 'infos': infos}


@view_config(route_name='home', request_method='POST',
renderer='templates/mytemplate.pt')

def contact_post(request):

    """Process contact datas"""
```

```
email, subject_id, text = map(request.POST.get, ('email',  
'subject_id', 'text'))
```

```
with transaction.manager:
```

```
    DBSession.add(Contact(email=email, subject_id=subject_id,  
text=text))
```

```
request.session.flash("Your submission has been registered", 'infos')
```

```
return HTTPFound(location=request.route_url('home'))
```

En este último ejemplo, cabe destacar bastantes cosas. La primera es la firma de cada función, que no recibe como parámetro más que la consulta. Este objeto consulta resulta esencial y conviene aprender a utilizarlo, por ejemplo para recuperar los datos.

A continuación se utiliza el decorador y el ORM, tal y como hemos visto anteriormente.

Por último, cabe destacar el uso de una sesión. Se utiliza para visualizar que el formulario se ha procesado correctamente. En efecto, el formulario se procesa en dos etapas. En primer lugar, cuando se hace clic en el botón de envío, los datos se envían al controlador, que los procesa, y a continuación realiza una redirección al controlador encargado de mostrar de nuevo el formulario. Es preciso que ambos controladores se comuniquen.

Para hacer funcionar esta sesión, hay que implementarla, lo que se realiza en el archivo **__init__** (las modificaciones se indican en negrita):

```
from pyramid.config import Configurator
```

```
from sqlalchemy import engine_from_config
```

```
from pyramid.session import UnencryptedCookieSessionFactoryConfig
```

```
from .models import (
```

```
    DBSession,
```

```
    Base,
```

```
)
```



```
def main(global_config, **settings):

    """ This function returns a Pyramid WSGI application.
    """

    engine = engine_from_config(settings, 'sqlalchemy.')

    DBSession.configure(bind=engine)

    Base.metadata.bind = engine

    # Session Configuration

    my_session_factory =

UnencryptedCookieSessionFactoryConfig('session_key_generator')

    config = Configurator(settings=settings,

session_factory=my_session_factory)

    # config = Configurator(settings=settings)

    config.include('pyramid_chameleon')

    config.add_static_view('static', 'static', cache_max_age=3600)

    config.add_route('home', '/')

    config.scan()

    return config.make_wsgi_app()
```

Habría que agregar muchas cosas, como por ejemplo la manera de gestionar la autenticación y los permisos, lo que parece esencial en la mayoría de los sitios; pero en este punto se obtiene un resultado que ya es visible y nos da una buena idea de las posibilidades de Pyramid.

Acaba de terminar la creación del primer sitio de Internet, con dos controladores, una vista y dos modelos. Acaba de implementar también el entorno y ha podido conocer aquellos aspectos esenciales de él que le conviene dominar, todo en 30 minutos, escribiendo apenas algunas pocas líneas de código fuente. Le invito, a continuación, a seguir inspyration.org y su repositorio de código fuente, el sitio vinculado a este libro. Mis proyectos libres, algunos de ellos realizados con Pyramid, están destinados a enriquecer este libro con nociones mucho más avanzadas. Encontrará también algunos tutoriales.

He aquí una captura de pantalla del resultado, tras enviar el formulario:

Para ir más allá

Se han presentado aquí los fundamentos que permiten crear un sitio web a partir de un framework minimalista en Python 3. Haría falta un libro entero dedicado a este tema, pues las competencias necesarias son enormes: Pyramid permite hacer muchas más cosas que las que se han presentado aquí y requiere una verdadera inversión de tiempo, sin hablar de las tecnologías HTML, CSS, JavaScript, asociadas a perfiles de diseñadores gráficos y de ergonomía.

Esta presentación puede, también, completarse con otros temas potencialmente complejos como, por ejemplo, la gestión correcta de los usuarios y de los grupos, la autenticación mediante LDAP, los grupos LDAP, la gestión de openID, el uso de varias bases de datos o la implementación de herramientas de profiling.

Por último, para finalizar, cuando se pone en producción una aplicación de estas características, conviene utilizar un servidor como Apache, Unicorn, Tornado o NGINX y, por tanto, WSGI. Esto se lleva a cabo muy rápidamente y de manera sencilla; gracias a los hosts virtuales es posible integrar esta aplicación en un pool de aplicaciones que se ejecuten en un mismo servidor.

Cabe tener en cuenta que Pyramid no es la única solución y que para realizar ciertas funcionalidades es necesario invocar a otros módulos Python, no todos ellos necesariamente migrados a Python 3, lo cual puede suponer un problema. Pyramid se sitúa como un framework web sobre una base de datos relacional, pero que puede trabajar con otros tipos de bases de datos y su tamaño, modesto, lo convierte en un buen candidato para adaptarse a situaciones particulares.

Existen algunos competidores muy apreciados. Uno de los más populares es Django, que tiene exactamente la misma posición que Pyramid. La diferencia entre ambos es que Django posee sus propios componentes para resolver todas las problemáticas. De este modo, es menos flexible pero más coherente, y aprovecha una documentación también muy buena.

Antes de concluir este capítulo, debemos hablar de Twisted. Este framework es muy particular y no es un framework MVC. Se trata de un framework que permite realizar una programación orientada a eventos y que permite resolver problemáticas de red. Si bien no tiene, a priori, nada que ver con todo lo que hemos expuesto en este capítulo, en realidad es un complemento que puede resultar muy útil. No obstante, es muy especializado y difícil de dominar.

Crear una aplicación Web

El objetivo de este capítulo es presentar una aplicación de consola basada en el trabajo realizado en el capítulo anterior, pues vamos a reutilizar la base de datos creada, así como toda la parte correspondiente al modelo.

El objetivo es iniciar un programa que permita introducir un contacto de forma rápida, escribiendo los argumentos directamente por línea de comandos.

A diferencia de un sitio web, una aplicación de consola tiene una interacción limitada con el usuario. No existe una interfaz de usuario avanzada, ni un formulario para presentar. Se contenta con recibir los parámetros y mostrar un mensaje en caso de error.

Crearemos esta aplicación de la manera más sencilla posible, es decir, reutilizando el modelo ya existente y dotando de seguridad a los datos introducidos mediante un módulo `argparse`. Como puede imaginarse, en una segunda etapa, es posible utilizar la entrada estándar para escribir los datos, o crear un programa para mostrar los contactos existentes, por ejemplo, aunque esto se sale del marco y el alcance de este capítulo.

Llamaremos a nuestra aplicación de consola con los siguientes parámetros:

```
$ contacto desarrollo.ini yo@casa.org Python "Mi mensaje"
```

El primer argumento designa el archivo de configuración de la aplicación. A continuación, se escribe el correo electrónico, el asunto y el mensaje.

Registrar el script

La primera tarea consiste en registrar esta nueva aplicación. Para ello, hay que modificar el archivo `setup.py`, agregando la siguiente línea en negrita:

```
setup(name='contacto',
      version='0.0',
      description='contacto',
      [...],
      install_requires=requires,
      entry_points="""\
      [paste.app_factory]
      main = contacto:main
      [console_scripts]
      initialize_contacto_db = contacto.scripts.initializedb:main
      show_settings = contacto.scripts.settings:main
      contacto = contacto.scripts.contacto:main
      """,
      )
```

De este modo se agrega a la aplicación un nuevo punto de entrada, que describe la función `main` del módulo `contacto`, situado en la carpeta `script` (junto al script de inicialización de la base de datos). A continuación es necesario redespigar la aplicación (en un entorno virtual):

```
$ python setup.py desarrollo
```

Creación de los datos

A continuación, es preciso crear el archivo `contacto.py` y agregar la función principal, que realiza el trabajo:

```
def contacto(DBSession, email, subject, text):
    """Agregar un contacto"""
    obj = DBSession.query(Subject).filter_by(name=subject).first()
    if not obj:
        print('Seleccione un asunto de la lista:')
        for obj in DBSession.query(Subject).all():
```

```
        print('> %s' % obj.name)
    print('Prueba de nuevo.')
    return
    with transaction.manager:
        DBSession.add(Contact(email=email, subject_id=obj.id, text=text))
```

Esta función necesita una sesión funcional y tres argumentos útiles para crear el dato.

Es importante destacar que, para una aplicación de consola sencilla, no existe ningún equivalente a una lista de selección como en la Web. De este modo, es posible seleccionar el asunto, escribiéndolo y buscándolo a continuación en la base de datos. Si no existe, entonces no se inserta el dato y se muestra un mensaje de error. También podríamos haber optado por agregarlo, si no existe, aunque hemos preferido mantener un comportamiento similar al de la Web.

Si el programa falla, el usuario puede invocar al comando mediante el histórico y modificar únicamente el asunto para seleccionar uno correcto.

Observe que también es posible utilizar un cursor, aunque en este caso superaríamos los diez minutos de desarrollo, y estamos viendo una sencilla introducción.

Ahora que todo está preparado, simplemente queda crear el parser de argumentos. Este se encargará de invocar a nuestra función, aunque también realizará ciertos controles de integridad sobre los argumentos, lo cual es muy importante para evitar inserciones incorrectas en la base de datos.

Parser de argumentos

Hay que crear un simple parser que se encargue de crear un contacto.

He aquí dicho parser:

```
def get_parser():
    # Etapa 1: definir una función proxy hacia la función principal
    def proxy_contacto(args):
        """Función proxy hacia contacto"""
        try:
            env = bootstrap(args.config_uri)
        except:
            print('Configuration file is not valid: %s' % args.config_uri)
            return
        settings, closer = env['registry'].settings, env['closer']
        try:
            engine = engine_from_config(settings, 'sqlalchemy.')
            DBSession.configure(bind=engine)
            Base.metadata.bind = engine
            contact(DBSession, args.email, args.subject, args.text)
        finally:
            closer()

    # Etapa 2: definir el analizador general
    parser = argparse.ArgumentParser(
        prog = 'contacto',
        description = """Programa que permite agregar un Contacto""",
        epilog = """Realizado para el libro Python, los fundamentos
```

```
del lenguaje"""
)
# Agregar opciones útiles
parser.add_argument(
    'config_uri',
    help = """archivo de configuración""",
    type = str,
)
parser.add_argument(
    'email',
    help = """Dirección de correo electrónico""",
    type = str,
)
parser.add_argument(
    'subject',
    help = """Asunto""",
    type = str,
)
parser.add_argument(
    'text',
    help = """Mensaje""",
    type = str,
)

# Etapa 3: agregar el vínculo entre el analizador y la función proxy
para calcula_capital
parser.set_defaults(func=proxy_contact)
return parser
```

En este ejemplo, se han visto las etapas clásicas: crear el parser, agregar las opciones detallándolas y vincularlo a una función proxy.

La parte importante es, en realidad, esta función proxy. Es ella la encargada de leer el archivo de configuración, extraer los parámetros mediante la función bootstrap, establecer el entorno y volver a cerrarlo.

Este procedimiento es similar a lo que se ha visto en el capítulo anterior para probar mediante la consola la base de datos.

Con este parser terminamos este capítulo y, como hemos podido constatar, hemos sido capaces de crear aquí una aplicación de consola en menos de diez minutos y con apenas unas cincuenta líneas de código.

Crear una Aplicación en Consola

El objetivo de este capítulo es presentar una aplicación de consola basada en el trabajo realizado en el capítulo anterior, pues vamos a reutilizar la base de datos creada, así como toda la parte correspondiente al modelo.

El objetivo es iniciar un programa que permita introducir un contacto de forma rápida, escribiendo los argumentos directamente por línea de comandos.

A diferencia de un sitio web, una aplicación de consola tiene una interacción limitada con el usuario. No existe una interfaz de usuario avanzada, ni un formulario para presentar. Se contenta con recibir los parámetros y mostrar un mensaje en caso de error.

Crearemos esta aplicación de la manera más sencilla posible, es decir, reutilizando el modelo ya existente y dotando de seguridad a los datos introducidos mediante un módulo `argparse`. Como puede imaginarse, en una segunda etapa, es posible utilizar la entrada estándar para escribir los datos, o crear un programa para mostrar los contactos existentes, por ejemplo, aunque esto se sale del marco y el alcance de este capítulo.

Llamaremos a nuestra aplicación de consola con los siguientes parámetros:

```
$ contacto desarrollo.ini yo@casa.org Python "Mi mensaje"
```

El primer argumento designa el archivo de configuración de la aplicación. A continuación, se escribe el correo electrónico, el asunto y el mensaje.

Registrar el script

La primera tarea consiste en registrar esta nueva aplicación. Para ello, hay que modificar el archivo `setup.py`, agregando la siguiente línea en negrita:

```
setup(name='contacto',
      version='0.0',
      description='contacto',
      [...],
      install_requires=requires,
      entry_points="""\
      [paste.app_factory]
      main = contacto:main
      [console_scripts]
      initialize_contacto_db = contacto.scripts.initializedb:main
      show_settings = contacto.scripts.settings:main
      contacto = contacto.scripts.contacto:main
      """,
      )
```

De este modo se agrega a la aplicación un nuevo punto de entrada, que describe la función `main` del módulo `contacto`, situado en la carpeta `script` (junto al script de inicialización de la base de datos). A continuación es necesario redespigar la aplicación (en un entorno virtual):

```
$ python setup.py desarrollo
```

Creación de los datos

A continuación, es preciso crear el archivo `contacto.py` y agregar la función principal, que realiza el trabajo:

```
def contacto(DBSession, email, subject, text):
    """Agregar un contacto"""
    obj = DBSession.query(Subject).filter_by(name=subject).first()
    if not obj:
        print('Seleccione un asunto de la lista:')
        for obj in DBSession.query(Subject).all():
```

```
        print('> %s' % obj.name)
    print('Pruebe de nuevo.')
    return
    with transaction.manager:
        DBSession.add(Contact(email=email, subject_id=obj.id, text=text))
```

Esta función necesita una sesión funcional y tres argumentos útiles para crear el dato.

Es importante destacar que, para una aplicación de consola sencilla, no existe ningún equivalente a una lista de selección como en la Web. De este modo, es posible seleccionar el asunto, escribiéndolo y buscándolo a continuación en la base de datos. Si no existe, entonces no se inserta el dato y se muestra un mensaje de error. También podríamos haber optado por agregarlo, si no existe, aunque hemos preferido mantener un comportamiento similar al de la Web.

Si el programa falla, el usuario puede invocar al comando mediante el histórico y modificar únicamente el asunto para seleccionar uno correcto.

Observe que también es posible utilizar un cursor, aunque en este caso superaríamos los diez minutos de desarrollo, y estamos viendo una sencilla introducción.

Ahora que todo está preparado, simplemente queda crear el parser de argumentos. Este se encargará de invocar a nuestra función, aunque también realizará ciertos controles de integridad sobre los argumentos, lo cual es muy importante para evitar inserciones incorrectas en la base de datos.

Hay que crear un simple parser que se encargue de crear un contacto.

He aquí dicho parser:

```
def get_parser():
    # Etapa 1: definir una función proxy hacia la función principal
    def proxy_contacto(args):
        """Función proxy hacia contacto"""
        try:
            env = bootstrap(args.config_uri)
        except:
            print('Configuration file is not valid: %s' % args.config_uri)
            return
        settings, closer = env['registry'].settings, env['closer']
        try:
            engine = engine_from_config(settings, 'sqlalchemy.')
            DBSession.configure(bind=engine)
            Base.metadata.bind = engine
            contact(DBSession, args.email, args.subject, args.text)
        finally:
            closer()

    # Etapa 2: definir el analizador general
    parser = argparse.ArgumentParser(
        prog = 'contacto',
        description = """Programa que permite agregar un Contacto""",
        epilog = """Realizado para el libro Python, los fundamentos
del lenguaje"""
    )
    # Agregar opciones útiles
    parser.add_argument(
        'config_uri',
```

```
        help = """"archivo de configuración""",
        type = str,
    )
    parser.add_argument(
        'email',
        help = """"Dirección de correo electrónico""",
        type = str,
    )
    parser.add_argument(
        'subject',
        help = """"Asunto""",
        type = str,
    )
    parser.add_argument(
        'text',
        help = """"Mensaje""",
        type = str,
    )
    )

# Etapa 3: agregar el vínculo entre el analizador y la función proxy
para calcula_capital
    parser.set_defaults(func=proxy_contact)
    return parser
```

En este ejemplo, se han visto las etapas clásicas: crear el parser, agregar las opciones detallándolas y vincularlo a una función proxy.

La parte importante es, en realidad, esta función proxy. Es ella la encargada de leer el archivo de configuración, extraer los parámetros mediante la función bootstrap, establecer el entorno y volver a cerrarlo.

Este procedimiento es similar a lo que se ha visto en el capítulo anterior para probar mediante la consola la base de datos.

Con este parser terminamos este capítulo y, como hemos podido constatar, hemos sido capaces de crear aquí una aplicación de consola en menos de diez minutos y con apenas unas cincuenta líneas de código.

Crear una Aplicación Gráfica

Breve presentación de Gtk y algunos trucos

1. Presentación

Gtk es una librería gráfica asociada al entorno gráfico Gnome. Forma parte del proyecto Gimp: sus creadores pensaron que era una lástima haber creado tantos componentes sin ofrecer la oportunidad a los demás de utilizar su formidable trabajo para crear otras aplicaciones.

PyGTK es una librería gráfica de libre distribución que conecta GTK+ a Python 2. Para Python 3, se trata de gi.repository. Esta última está escrita en C (utiliza Cairo) y forma parte de un conjunto mucho más amplio. Se utiliza en numerosas aplicaciones de referencia, como Gimp, por ejemplo, que está en el germen de entornos de escritorio como Gnome y Xfce. Está vinculada a la Libglade y a una aplicación de creación de interfaces gráficas Glade que es muy práctica y eficaz.

La librería GTK+ ha sido objeto de profundos cambios que han dado como resultado la aparición de GTK3+ (con rupturas de la compatibilidad hacia atrás perfectamente comprensibles) y en Python 3 disponemos únicamente de esta nueva rama.

Es importante destacar la existencia de una documentación de referencia indispensable que permite arrancar progresivamente, paso a paso, y que le aconsejo revisar antes o después de leer este capítulo: <http://readthedocs.org/docs/python-gtk-3-tutorial/en/latest/index.html>

En efecto, el ejemplo que explicamos aquí complementa esta documentación, presentando otros aspectos.

Los dos puntos importantes abordados aquí son la separación de los distintos elementos del código y el uso de la herramienta **Glade** para crear interfaces de ratón.

2. Trucos

Se va a reutilizar parte del trabajo que ya se ha realizado durante la construcción de una aplicación de consola. De este modo gestionaremos, de manera muy sencilla, el hecho de disponer de un comando que recibe como parámetro el archivo de configuración adecuado para extraer los parámetros útiles, con el objetivo de crear la sesión, cuyo objeto central nos permite crear nuestros datos.

A continuación crearemos una interfaz muy sencilla, mediante la herramienta **Glade**, para diseñar, con algunos pocos clics de ratón, los tres campos que necesitaremos, así como los controles útiles.

Por último, crearemos un controlador encargado de lanzar la interfaz gráfica y que posea los métodos necesarios para utilizar el modelo con el objetivo de leer o escribir los datos.

El aspecto importante que es preciso tener en cuenta es que no deben mezclarse las cosas. No sería práctico implementarlo todo en la interfaz gráfica. En efecto, no es una clase concebida para hacer que **Gtk** juegue con la sesión **SQLAlchemy**.

Esta clase debe contentarse con ser un simple vínculo y debe concentrarse en lo que se denomina la experiencia de usuario, es decir, describir lo que debe ocurrir cuando el usuario hace clic en un campo, sobre un botón o cuando pasa por encima de una etiqueta...

La base de las librerías gráficas tales como Gtk es el uso de eventos. Cada vez que ocurre algo (movimiento del ratón, escritura mediante teclado, paso por encima de un campo concreto...), se generan eventos.

La mayor parte del tiempo, no ocurre nada, dado que no hay nadie encargado de leer dichos eventos. Basta con vincular un evento a un método para que dicho método se invoque cuando se produzca el evento.

De manera indirecta, estamos utilizando aquí varios patrones de diseño, entre ellos la llamada de retorno o callback.

Uno de los ejemplos habituales es el botón. Su clic puede asociarse a un evento mediante dicho callback. La dificultad principal consiste en crear varios botones que deban utilizar el mismo método callback, pero de manera distinta.

Encontrará en la documentación oficial ejemplos que le permitirán crear un callback para una función que pueda utilizarse por un único botón. Si desea pasarle parámetros, es necesario utilizar un objeto al que se le puedan pasar parámetros y cuyo resultado sea una función. En Python, una función no es más que un objeto que posee un método `__call__`.

Se crea, por tanto, un objeto de este tipo:

```
class Ejemplo:
    def __init__(self, callback, **kw):
        self.callback = callback
        self.kw = kw
    def __call__(self):
        self.callback(**self.kw)
```

Esta manera de utilizar las funciones resulta esencial en la programación de interfaces gráficas.

Es, también, importante señalar otro truco. **Gtk** no es fácil de instalar en un entorno virtual. Para ello, basta con instalarlo en el entorno del sistema mediante el gestor de paquetes y, a continuación, agregar el paquete al entorno virtual, estableciendo un vínculo:

```
$ ln -s /usr/lib/python3/dist-packages/gi ../lib/python3.2/site-packages/
```

Iniciar el programa

Como hemos visto anteriormente, es preciso crear un nuevo punto de entrada para arrancar nuestro programa, en el archivo `setup.py`:

```
setup(name='contact',
      version='0.0',
      description='contact',
      [...],
      install_requires=requires,
      entry_points="""\
[paste.app_factory]
main = contact:main
[console_scripts]
initialize_contact_db = contact.scripts.initializedb:main
show_settings = contact.scripts.settings:main
contact = contact.scripts.contact:main          gcontact =
contact.scripts.gtk:main
""",
      )
```

A continuación hay que crear un archivo `gtk.py` en la carpeta `scripts`, donde ya podemos agregar código:

```
#!/usr/bin/python3

import argparse

from pyramid.paster import bootstrap
from sqlalchemy import engine_from_config
```

```
from contact.models import DBSession, Base, Contact, Subject

import transaction
class Controller:
    def __init__(self, DBSession):
        self.DBSession = DBSession
        print('This test is a success')

def get_parser():
    # Etapa 1: definir una función proxy hacia la función principal
    def proxy_gcontact(args):
        """Función proxy hacia contact"""
        try:
            env = bootstrap(args.config_uri)
        except:
            print('Configuration file is not valid: %s' % args.config_uri)
            return
        settings, closer = env['registry'].settings, env['closer']
        try:
            engine = engine_from_config(settings, 'sqlalchemy.')
            DBSession.configure(bind=engine)
            Base.metadata.bind = engine
            Controller(DBSession)
        finally:
            closer()

    # Etapa 2: definir el analizador general
    parser = argparse.ArgumentParser(
        prog = 'contact',
        description = """Programa que permite agregar un Contacto""",
        epilog = """Realizado por el libro Python, los fundamentos
del lenguaje"""
    )

    # Agregar las opciones útiles

    parser.add_argument(
        'config_uri',
        help = """archivo de configuración""",
        type = str,
    )

    # Etapa 3: agregar el vínculo entre el analizador y la función proxy
    parser.set_defaults(func=proxy_gcontact)
    return parser


def main():
    parser = get_parser()
    # Etapa 4: iniciar el análisis de argumentos, y a continuación
    el programa.
    args = parser.parse_args()
    args.func(args)

# Fin del programa
```

Como puede comprobarse, se crea una clase controlador que agrega la sesión **SQLAlchemy** y que muestra un mensaje indicando que todo va bien. El resto es muy similar a lo expuesto en el capítulo anterior, a excepción de que se recibe un único argumento, en lugar de varios. Simplemente queda redespargar la aplicación para verificar que este esqueleto de script funciona:

```
$ python setup.py develop
```

Ahora, podemos probar nuestro nuevo comando y, simplemente, ver el mensaje que se muestra.

```
$ gcontact development.ini
```

Es posible, también, probar el método sin argumentos, o pasando demasiados argumentos, o incluso con un archivo de configuración incorrecto para comprobar que los errores se gestionan correctamente a este nivel.

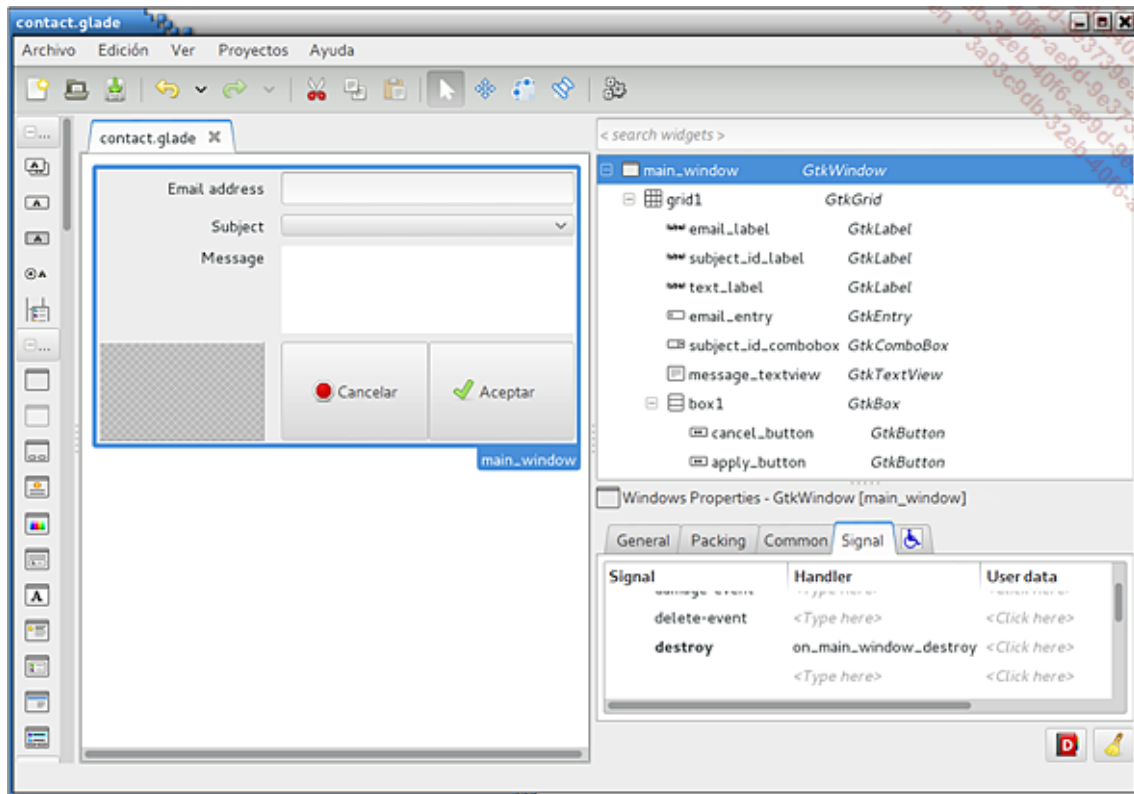
nterfaz gráfica con Glade

Cuando se trabaja con **Gtk**, la adopción de una herramienta potente como **Glade** supone un verdadero ahorro de tiempo. Por otro lado, el resultado producido puede utilizarse tanto en Python como en C.

En este ejemplo, se ha diseñado una ventana, llamada `main_window`, que contiene una malla de dos columnas y cuatro líneas. Las tres primeras líneas de la primera columna contienen etiquetas.

Puede observar que el nombre asignado a cada uno de los componentes es importante y puede modificarse.

En la segunda columna, se ha agregado, sucesivamente, un campo de entrada (Entry), una lista de selección (ComboBox), una zona de texto (TextView) y, por último, un cuadro que contiene dos espacios que colocaremos de manera horizontal para agregar dos botones.



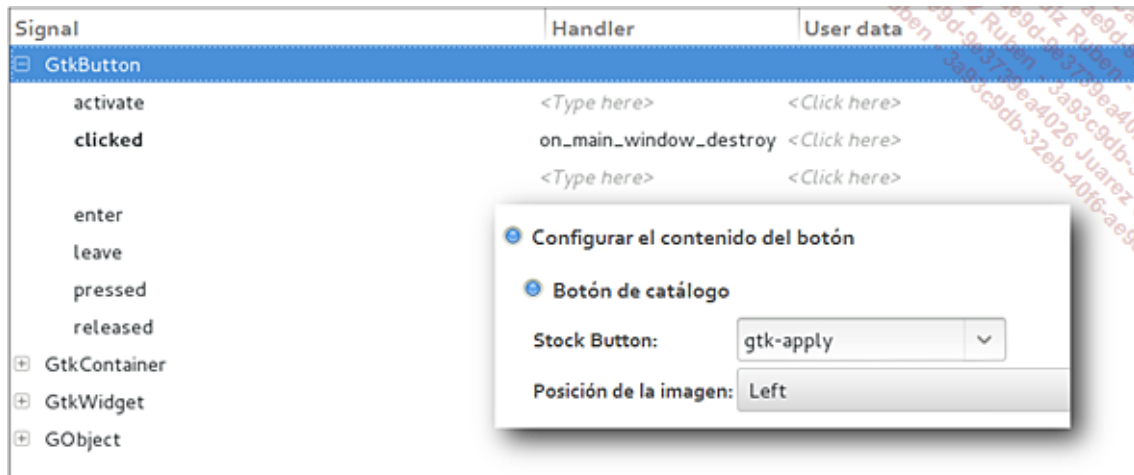
Interfaz general de **Glade** y eventos en la ventana

Como puede comprobar, la interfaz no es, necesariamente, agradable a la vista. Pero lo que importa son los parámetros que encontrará en las distintas pestañas del panel situado abajo a la derecha.

Puede hacer que los distintos elementos de la malla tengan la misma longitud o no, que estén centrados, alineados en la parte superior derecha...

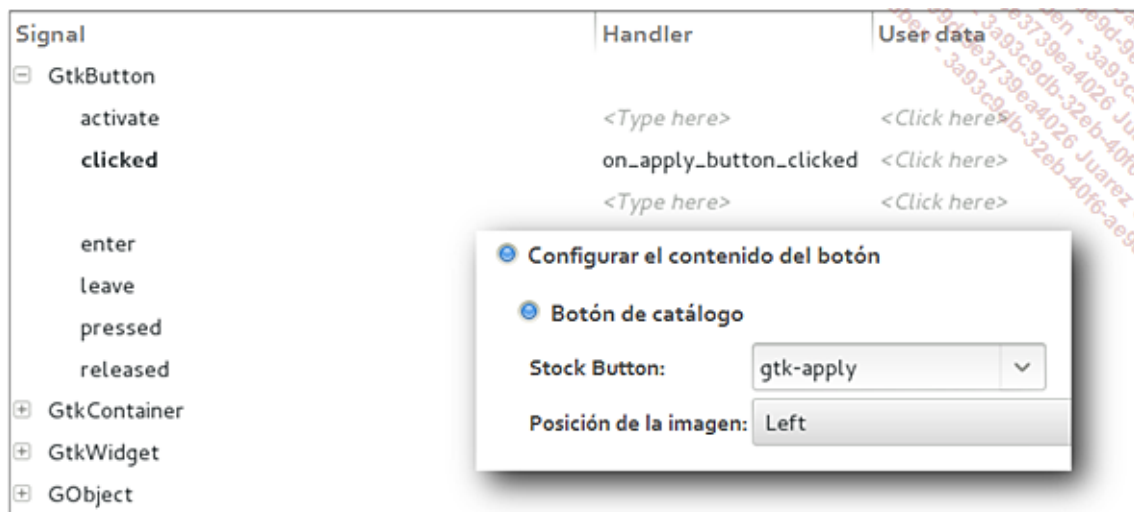
Todavía más importante, puede seleccionar los callback que desea vincular a los eventos particulares. En la ventana principal, el evento importante es el clic en la cruz que permite salir de la aplicación. Aquí, se asocia al método llamado `on_main_window_destroy`.

Los otros dos elementos que queremos colocar rápidamente son los botones. Se va a vincular el botón de anulación con el mismo evento, que nos permite salir de la aplicación si se hace clic en él. A continuación, conviene homogeneizar su rol parametrizándolo como se indica a continuación:



Parametrización del botón de anulación

Queda por parametrizar el botón de validación:

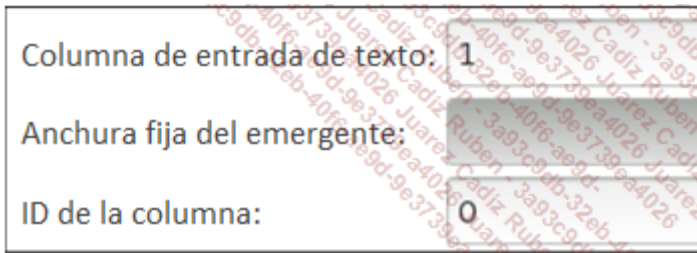


Parametrización del botón de validación

Para todos los demás elementos, puede revisar los distintos parámetros propuestos, realizar pruebas y darse cuenta usted mismo de las consecuencias que se obtienen modificando sus atributos. No obstante, el uso de uno de los componentes no es evidente a primera vista. Se trata de la lista de selección.

En efecto, es preciso completar la lista con los valores adecuados. Esta lista muestra el texto del asunto, pero devolverá su identificador. Sin anticipar lo que viene a continuación, admitamos aquí que se utilizará una simple 2-tupla.

El primer elemento será el identificador y el segundo el texto. Es preciso indicarlo en la interfaz **Glade**:



Columna de entrada de texto: 1

Anchura fija del emergente: 0

ID de la columna: 0

Parametrización de la lista de selección

Disponemos ahora de todos los objetos necesarios. El interés de utilizar Glade, además del hecho de que se gana en rapidez, es también poder disponer en una interfaz clara y legible del conjunto de opciones, sin tener que acudir a la documentación. Dicho de otro modo, a primera vista, existen muchos parámetros, aunque hay que saber que la mayoría de ellos no son útiles más que en situaciones muy concretas y que, si no sabemos para qué sirve algún parámetro, basta con dejar su valor por defecto. En otro caso, podemos simplemente modificar su valor y reiniciar la aplicación para comprobar el cambio.

Para encontrar en la documentación el significado de ciertos parámetros, conviene revisar la documentación de GTK+, es decir, de la librería original.

Guardamos este archivo en la carpeta templates, ya que diseña, en cierto modo, una interfaz de usuario.

Crear el componente gráfico

El componente gráfico dispone de tres métodos: un método de inicialización y dos métodos de callback, es decir, el método para salir de la aplicación cuando se hace clic en la cruz o en el botón de anulación, y el método de validación que creará un contacto.

He aquí la clase, con las correspondientes explicaciones para cada etapa:

```
class GtkContact:
    def __init__(self, controller):
        self.controller = controller
```

Empezamos agregando el controlador a la instancia en curso, lo que dará acceso a los métodos útiles que permiten recuperar los asuntos o crear el contacto. A continuación, es preciso cargar la interfaz que acabamos de diseñar con Glade:

```
interface = Gtk.Builder()
interface.add_from_file('contact/templates/contact.glade')
```

Nos contentamos con crear el builder y cargar el archivo **Glade**. A continuación, hay que recuperar punteros hacia los campos útiles, tarea que podemos realizar porque se han nombrado correctamente en Glade:

```
# Vínculos hacia los campos útiles
self.email = interface.get_object("email_entry")
self.subject = interface.get_object("subject_id_combobox")
self.text = interface.get_object("message_textview")
```

Ahora, es posible atacar al encargado de rellenar la lista de selección. Esto se realiza en varias etapas. En primer lugar es preciso crear un objeto destinado a agregar los datos:

```
store = Gtk.ListStore(int, str)
for subject in controller.get_subjects():
    store.append([subject.id, subject.name])
```

Cabe destacar que este mismo componente se utiliza en los árboles de datos o bien para las tablas. Se trata de un elemento muy importante.

A continuación, hay que vincularlo con nuestra lista de selección, y utilizar un objeto particular para hacer aparecer el texto:

```
self.subject.set_model(store)
cell = Gtk.CellRendererText()
self.subject.pack_start(cell, True)
self.subject.add_attribute(cell, "text", 1)
```

Una vez realizado, vinculamos los métodos de la clase en curso con los eventos declarados en **Glade**:

```
interface.connect_signals(self)
```

Y mostramos la ventana para que el usuario pueda verla:

```
window = interface.get_object("main_window")
window.show_all()
```

Ahora, tan solo queda crear los dos callback, tal y como hemos hecho para vincular los componentes diseñados con **Glade**. Poseen, efectivamente, el mismo nombre que el declarado en **Glade**:

```
def on_main_window_destroy(self, widget):
    Gtk.main_quit()
```

Para el primer evento, se trata de cortar el bucle de eventos y terminar el programa. Para la otra función, se recuperan los datos introducidos por el usuario:

```
def on_apply_button_clicked(self, widget):
    # Recuperar del valor de un campo de texto
    email = self.email.get_text()
```

Si bien resulta sencillo para un campo de texto, existen más etapas para gestionar la lista de selección. Es posible recuperar la pareja de datos:

```
# Recuperar el valor de la lista desplegable
tree_iter = self.subject.get_active_iter()
if tree_iter is None:
    return
model = self.subject.get_model()
row_id, name = model[tree_iter][:2]
```

El procedimiento es bastante estándar. En este caso, nos interesa el identificador:

```
subject_id = row_id
```

A continuación se muestra cómo recuperar la integridad del texto informado en una zona de texto:

```
# Recuperar el valor de un campo de texto multilínea
message = self.text.get_buffer()
text = message.get_text(message.get_start_iter(),
message.get_end_iter(), False)
```


Ahora que la parte esencial del trabajo está realizada, podemos crear el contacto:

```
# Crear el contacto
self.controller.add_contact(email, subject_id, text)
```

Para controlar realmente **Gtk**, hay que pasar tiempo probando cada uno de los componentes con el apoyo de un tutorial. Se verá rápidamente que se trata de los mismos principios a los que estamos acostumbrados y, de este modo, podremos adquirir los conocimientos mínimos necesarios.

Controlador

La interfaz tiene como objetivo permitir al usuario introducir un contacto. Debe, también, proveer la lista de asuntos, de modo que pueda recuperarse a partir de una lista de selección. Se trata de las dos únicas acciones que esperamos por parte del controlador. Es también el encargado de generar la interfaz gráfica y ejecutar el bucle de eventos.

He aquí el controlador:

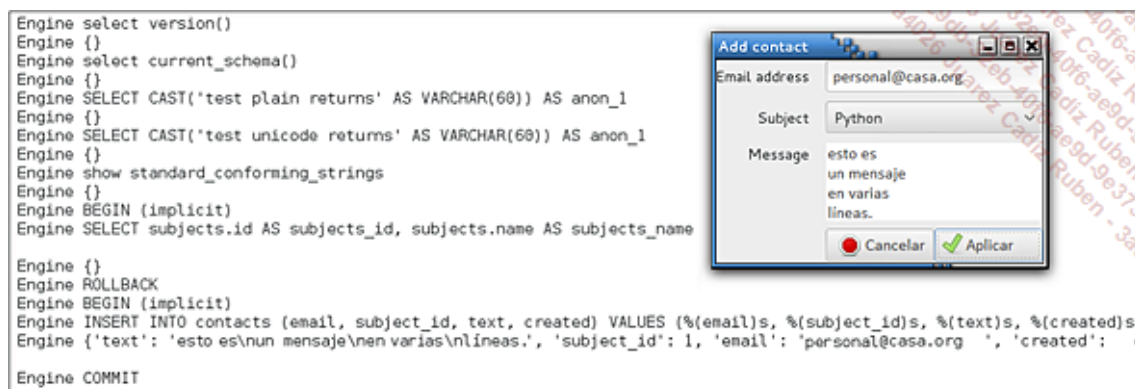
```
class Controller:
    def __init__(self, DBSession):
        self.DBSession = DBSession
        GtkContact(self)
        Gtk.main()

    def get_subjects(self):
        return DBSession.query(Subject).all()

    def add_contact(self, email, subject_id, text):
        with transaction.manager:
            DBSession.add(Contact(email=email, subject_id=subject_id,
text=text))
```

Tras la llamada a `Gtk.main`, el programa se pone en pausa y a la escucha de eventos generados por el usuario. Cuando se sale de la aplicación, invocando a `Gtk.main_quit`, el bucle termina.

He aquí el resultado:



Aplicación gráfica con el log de SQLAlchemy de fondo

Otras librerías gráficas

1. TkInter

TkInter es la librería gráfica utilizada por Python y es una migración de la librería gráfica de **Tk** creada para el lenguaje **TLC**.

Como gran ventaja cabe destacar que es fácil de implementar y se parece bastante a Gtk en su enfoque general.

2. wxPython

La librería **wxPython** es una librería que conecta los **wxWidgets** a Python. Es una alternativa a **TkInter** muy valorada y particularmente completa, próxima al sistema operativo.

La librería **wxWidgets** es una librería gráfica de libre distribución escrita en C++ que permite escribir una interfaz gráfica que será idéntica sea cual sea el sistema operativo, pero tomando su apariencia.

Esta librería es una referencia, y funciona en Python 3 a partir de la versión 3.3.

3. PyQt

PyQt es una librería gráfica de libre distribución que conecta **Qt** a Python. **Qt**, que debe pronunciarse como la palabra inglesa «cute», se escribe en C++ y es muy portable. El entorno gráfico KDE está construido en **Qt**. Muchos módulos complementarios adicionales hacen de ella mucho más que una simple librería que permite realizar interfaces gráficas.

Qt está vinculada a **QtDesigner**, que es, de manera similar a **Glade**, una aplicación que permite crear interfaces gráficas.

Qt está completamente migrada a Python 3.x desde su versión 4.5, lo que hace de ella un candidato excelente. También es bastante madura y estable en su evolución.

4. PySide

PySide es una nueva implementación de **Qt** para Python. La principal diferencia es la licencia más permisiva (LGPL en lugar de GPL). Está menos madura que **Qt**, aunque presenta ventajas e inconvenientes similares.

5. Otras

Existen otras librerías, más confidenciales o específicas. Toda la información útil y los enlaces están aquí: <http://docs.python.org/3/faq/gui.html>

