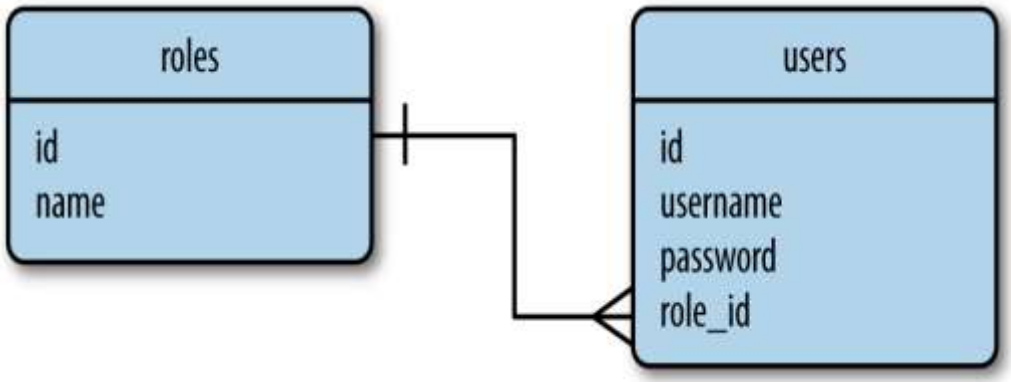
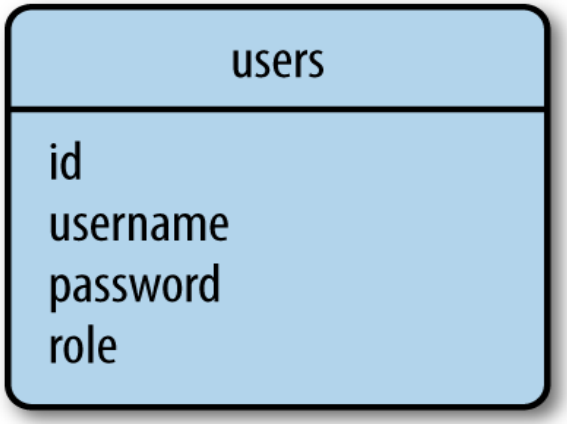
SQL Databases

Figure 5-1 shows a diagram of a simple database with two tables that store users and user roles. The line that connects the two tables represents a relationship between the tables.



NoSQL Databases

Databases that do not follow the relational model described in the previous section are collectively referred to as *NoSQL* databases. One common organization for NoSQL databases uses *collections* instead of tables and *documents* instead of records. NoSQL databases are designed in a way that makes joins difficult, so most of them do not support this operation at all.



SQL or NoSQL?

- SQL databases excel at storing structured data in an efficient and compact form. These databases go to great lengths to preserve consistency, even in the face of power failures or hardware malfunctions. The paradigm that allows relational databases to reach this high level of reliability is called ACID, which stands for Atomicity, Consistency, Isolation, and Durability.

- NoSQL databases relax some of the ACID requirements and as a result can sometimes get a performance edge.

For small to medium-sized applications, both SQL and NoSQL databases are perfectly capable and have practically equivalent performance.

Python Database Frameworks

Python has packages for most database engines, both open source and commercial. Flask puts no restrictions on what database packages can be used, so you can work with MySQL, Postgres, SQLite, Redis, MongoDB, CouchDB, or DynamoDB if any of these is your favorite.

Database Management with Flask-SQLAlchemy

Table 5-1. Flask-SQLAlchemy database URLs

Database engine	URL
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/database
SQLite (Linux, macOS)	sqlite:////absolute/path/to/database
SQLite	sqlite:///c:/absolute/path/to/database

Database Management with Flask-SQLAlchemy

Like most other extensions, Flask-SQLAlchemy is installed with *pip*:

```
(venv) $ pip install flask-sqlalchemy
```

Example 5-1. hello.py: database configuration

```
import os
from flask_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
  'sqlite:///' + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

The `db` object instantiated from the class `SQLAlchemy` represents the database and provides access to all the functionality of Flask-SQLAlchemy.

Model Definition

The database instance from Flask-SQLAlchemy provides a base class for models as well as a set of helper classes and functions that are used to define their structure. The `roles` and `users` tables from Figure 5-1 can be defined as the models `Role` and `User` as shown in Example 5-2.

Example 5-2. hello.py: Role and User model definition

```
class Role(db.Model):
  __tablename__ = 'roles'
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(64), unique=True)

  def __repr__(self):
    return '<Role %r>' % self.name

class User(db.Model):
  __tablename__ = 'users'
  id = db.Column(db.Integer, primary_key=True)
  username = db.Column(db.String(64), unique=True, index=True)

  def __repr__(self):
    return '<User %r>' % self.username
```

Table 5-2. Most common SQLAlchemy column types

Type name	Python type	Description
Integer	int	Regular integer, typically 32 bits Short-range integer, typically
SmallInteger	Int	16 bits
BigInteger	int or long	Unlimited precision integer
Float	float	Floating-point number
Numeric	decimal.Decimal	Fixed-point number
String	str	Variable-length string
Text	str	Variable-length string, optimized for large or unbounded length
Unicode	unicode	Variable-length Unicode string
UnicodeText	unicode	Variable-length Unicode string, optimized for large or unbounded length
Boolean	bool	Boolean value
Date	datetime.date	Date value
Time	datetime.time	Time value
DateTime	datetime.datetime	Date and time value
Interval	datetime.timedelta	Time interval
Enum	str	List of string values Automatic
PickleType	Any Python object	Pickle serialization Binary blob
LargeBinary	str	

The remaining arguments to `db.Column` specify configuration options for each attribute. Table 5-3 lists some of the options available.

Table 5-3. Most common SQLAlchemy column options

Option name	Description
`primary_key`	If set to `True`, the column is the table's primary key.
`unique`	If set to `True`, do not allow duplicate values for this column.
`index`	If set to `True`, create an index for this column, so that queries are more efficient.
`Nullable`	If set to `True`, allow empty values for this column. If set to `False`, the column will not allow null values.
`default`	Define a default value for the column.

Relationships

Relational databases establish connections between rows in different tables through the use of relationships. The relational diagram in Figure 5-1 expresses a simple rela- tionship between users and their roles. This is a *one-to-many* relationship from roles to users, because one role can belong to many users, but each user can have only one role.

Example 5-3 shows how the one-to-many relationship in Figure 5-1 is represented in the model classes.

Example 5-3. hello.py: relationships in the database models

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Table 5-4. Common SQLAlchemy relationship options

Option name	Description
`backref`	Add a back reference in the other model in the relationship.
`primaryjoin`	Specify the join condition between the two models explicitly. This is necessary only for ambiguous relationships.
`lazy`	Specify how the related items are to be loaded. Possible values are `select` (items are loaded on demand the first time they are accessed), `immediate` (items are loaded when the source object is loaded), `joined` (items are loaded immediately, but as a join), `subquery` (items are loaded immediately, but as a subquery), `noload` (items are never loaded), and `dynamic` (instead of loading the items, the query that can load them is given).
`uselist`	If set to `False`, use a scalar instead of a list.
`order_by`	Specify the ordering used for the items in the relationship.
`secondary`	Specify the name of the association table to use in many-to-many relationships.
`secondaryjoin`	Specify the secondary join condition for many-to-many relationships when SQLAlchemy cannot determine it on its own.

Database Operations

The following sections will walk you through the most common database operations in a shell started with the `flask shell` command. Before you use this command, make sure the `FLASK_APP` environment variable is set to `hello.py`, as shown in Chapter 2.

Creating the Tables

The very first thing to do is to instruct Flask-SQLAlchemy to create a database based on the model classes. The `db.create_all()` function locates all the subclasses of `db.Model` and creates corresponding tables in the database for them:

```
(venv) $ flask shell
>>> from hello import db
>>> db.create_all()
```

The brute-force solution to update existing database tables to a different schema is to remove the old tables first:

```
>>> db.drop_all()
>>> db.create_all()
```

Inserting Rows

The following example creates a few roles and users:

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

The objects exist only on the Python side so far; they have not been written to the database yet. Because of that, their `id` values have not yet been assigned:

```
>>> print(admin_role.id)
None
>>> print(mod_role.id)
None
>>> print(user_role.id)
None
```

Changes to the database are managed through a database *session*, which Flask-SQLAlchemy provides as `db.session`. To prepare objects to be written to the database, they must be added to the session:

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

Database sessions are also called *transactions*.

Or, more concisely:

```
>>> db.session.add_all([admin_role, mod_role, user_role,
...user_john, user_susan, user_david])
```

To write the objects to the database, the session needs to be *committed* by calling its `commit()` method:

```
>>> db.session.commit()
```

Check the `id` attributes again after having the data committed to see that they are now set:

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

> The `db.session` database session is not related to the Flask `session` object discussed in Chapter 4.

Modifying Rows

The `add()` method of the database session can also be used to update models. Con- tinuing in the same shell session, the following example renames the "Admin" role to "Administrator":

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

Deleting Rows

The database session also has a `delete()` method. The following example deletes the
"Moderator" role from the database:

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

Note that deletions, like insertions and updates, are executed only when the database session is committed.

Querying Rows

Flask-SQLAlchemy makes a `query` object available in each model class. The most basic query for a model is triggered with the `all()` method, which returns the entire contents of the corresponding table:

```
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>]
>>> User.query.all()
[<User 'john'>, <User 'susan'>, <User 'david'>]
```

A query object can be configured to issue more specific database searches through the use of *filters*. The following example finds all the users that were assigned the "User" role:

```
>>> User.query.filter_by(role=user_role).all()
[<User 'susan'>, <User 'david'>]
```

It is also possible to inspect the native SQL query that SQLAlchemy generates for a given query by converting the query object to a string:

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id \nFROM users \nWHERE :param_1 =
users.role_id'
```

The following example issues a query that loads the user role with name "User":

```
>>> user_role = Role.query.filter_by(name='User').first()
```

Table 5-5 shows some of the most common filters available to queries. The complete list is in the SQLAlchemy documentation.

Table 5-5. Common SQLAlchemy query filters

Option	Description
`filter()`	Returns a new query that adds an additional filter to the original query
`filter_by()`	Returns a new query that adds an additional equality filter to the original query
`limit()`	Returns a new query that limits the number of results of the original query to the given number
`offset()`	Returns a new query that applies an offset into the list of results of the original query
`order_by()`	Returns a new query that sorts the results of the original query according to the given criteria
`group_by()`	Returns a new query that groups the results of the original query according to the given criteria

Table 5-6. Most common SQLAlchemy query executors

Option	Description
`all()`	Returns all the results of a query as a list
`first()`	Returns the first result of a query, or `None` if there are no results
`first_or_404()`	Returns the first result of a query, or aborts the request and sends a 404 error as the response if there are no results
`get()`	Returns the row that matches the given primary key, or `None` if no matching row is found
`get_or_404()`	Returns the row that matches the given primary key or, if the key is not found, aborts the request and sends a 404 error as the response
`count()`	Returns the result count of the query
`paginate()`	Returns a `Pagination` object that contains the specified range of results

Relationships work similarly to queries. The following example queries the one-to- many relationship between roles and users from both ends:

```
>>> users = user_role.users
>>> users
[<User 'susan'>, <User 'david'>]
>>> users[0].role
<Role 'User'>
```

Database Use in View Functions

Example 5-4. hello.py: dynamic database relationships

```
class Role(db.Model):
  # ...
  users = db.relationship('User', backref='role', lazy='dynamic')
  # ...
```

With the relationship configured in this way, `user_role.users` returns a query that hasn't executed yet, so filters can be added to it:

```
>>> user_role.users.order_by(User.username).all()
[<User 'david'>, <User 'susan'>]
>>> user_role.users.count()
2
```

Example 5-5. hello.py: database use in view functions

```
@app.route('/', methods=['GET', 'POST'])
def index():
  form = NameForm()
  if form.validate_on_submit():
    user = User.query.filter_by(username=form.name.data).first()
    if user is None:
      user = User(username=form.name.data)
      db.session.add(user)   db.session.commit()
      session['known'] = False
    else:
      session['known'] = True   session['name'] =
    form.name.data   form.name.data = ''
    return redirect(url_for('index'))
  return render_template('index.html',   form=form,
    name=session.get('name'),
    known=session.get('known', False))
```

Integration with the Python Shell

Example 5-7. hello.py: adding a shell context

```
@app.shell_context_processor
def make_shell_context():
  return dict(db=db, User=User, Role=Role)
```

The shell context processor function returns a dictionary that includes the database instance and the models. The `flask shell` command will import these items automatically into the shell, in addition to `app`, which is imported by default:

```
$ flask shell
>>> app
<Flask 'hello'>
>>> db
<SQLAlchemy engine='sqlite:////home/flask/flasky/data.sqlite'>
>>> User
<class 'hello.User'>
```

Creating a Migration Repository

To begin, Flask-Migrate must be installed in the virtual environment:

```
(venv) $ pip install flask-migrate
```

Example 5-8. hello.py: Flask-Migrate initialization

```
from flask_migrate import Migrate

# ...

migrate = Migrate(app, db)
```

To expose the database migration commands, Flask-Migrate adds a `flask db` com- mand with several subcommands. When you work on a new project, you can add support for database migrations with the `init` subcommand:

```
(venv) $ flask db init
 Creating directory /home/flask/flasky/migrations...done
 Creating directory /home/flask/flasky/migrations/versions...done  Generating
 /home/flask/flasky/migrations/alembic.ini...done  Generating
 /home/flask/flasky/migrations/env.py...done  Generating
 /home/flask/flasky/migrations/env.pyc...done  Generating
 /home/flask/flasky/migrations/README...done  Generating
 /home/flask/flasky/migrations/script.py.mako...done  Please edit
 configuration/connection/logging settings in
 '/home/flask/flasky/migrations/alembic.ini' before proceeding.
```

Creating a Migration Script

To make changes to your database schema with Flask-Migrate, the following procedure needs to be followed:

1. Make the necessary changes to the model classes.
2. Create an automatic migration script with the `flask db migrate` command.
3. Review the generated script and adjust it so that it accurately represents the changes that were made to the models.
4. Add the migration script to source control.
5. Apply the migration to the database with the `flask db upgrade` command.

The `flask db migrate` subcommand creates an automatic migration script:

```
(venv) $ flask db migrate -m "initial migration"
    INFO  [alembic.migration] Context impl SQLiteImpl.
    INFO  [alembic.migration] Will assume non-transactional DDL.
    INFO  [alembic.autogenerate] Detected added table 'roles'
    INFO  [alembic.autogenerate] Detected added table 'users'
    INFO  [alembic.autogenerate.compare] Detected added index
'ix_users_username' on '['username']'
  Generating /home/flask/flasky/migrations/versions/1bc
  594146bb5_initial_migration.py...done
```

Upgrading the Database

Once a migration script has been reviewed and accepted, it can be applied to the data- base using the `flask db upgrade` command:

```
(venv) $ flask db upgrade

INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.migration] Running upgrade None -> 1bc594146bb5, initial migration
```

Adding More Migrations

The procedure to introduce a change in the database is similar to what was done to introduce the first migration:

1. Make the necessary changes in the database models.
2. Generate a migration with the `flask db migrate` command.
3. Review the generated migration script and correct it if it has any inaccuracies.
4. Apply the changes to the database with the `flask db upgrade` command.

While working on a specific feature, you may find that you need to make several changes to your database models before you get them the way you want them.

If your last migration has not been committed to source control yet, you can opt to expand it to incorporate new changes as you make them, and this will save you from having lots of very small migration scripts that are meaningless on their own. The procedure to expand the last migration script is as follows:

1. Remove the last migration from the database with the `flask db downgrade` com- mand (note that this may cause some data to be lost).
2. Delete the last migration script, which is now orphaned.
3. Generate a new database migration with the `flask db migrate` command, which will now include the changes in the migration script you just removed, plus any other changes you've made to the models.
4. Review and apply the migration script as described previously.

Email Support with Flask-Mail

Although the `smtplib` package from the Python standard library can be used to send email inside a Flask application, the Flask-Mail extension wraps `smtplib` and integrates it nicely with Flask. Flask-Mail is installed with *pip*:

```
(venv) $ pip install flask-mail
```

The extension connects to a Simple Mail Transfer Protocol (SMTP) server and passes emails to it for delivery. If no configuration is given, Flask-Mail connects to *localhost* at port 25 and sends email without authentication. Table 6-1 shows the list of configuration keys that can be used to configure the SMTP server.

Table 6-1. Flask-Mail SMTP server configuration keys

Key	Default	Description
MAIL_SERVER	*localhost*	Hostname or IP address of the email server
MAIL_PORT	25	Port of the email server
MAIL_USE_TLS	False	Enable Transport Layer Security (TLS) security
MAIL_USE_SSL	False	Enable Secure Sockets Layer (SSL) security
MAIL_USERNAME	None	Mail account username
MAIL_PASSWORD	None	Mail account password

Example 6-1. hello.py: Flask-Mail configuration for Gmail

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```

Example 6-2. hello.py: Flask-Mail initialization

```
from flask_mail import Mail

mail = Mail(app)
```

The two environment variables that hold the email server username and password need to be defined in the environment. If you are on Linux or macOS, you can set these variables as follows:

```
(venv) $ export MAIL_USERNAME=<Gmail username>
(venv) $ export MAIL_PASSWORD=<Gmail password>
```

For Microsoft Windows users, the environment variables are set as follows:

```
(venv) $ set MAIL_USERNAME=<Gmail username>
(venv) $ set MAIL_PASSWORD=<Gmail password>
```

Sending Email from the Python Shell

To test the configuration, you can start a shell session and send a test email

```
(venv) $ flask shell
 >>> from flask_mail import Message
 >>> from hello import mail
 >>> msg = Message('test email', sender='you@example.com',
 ...  recipients=['you@example.com'])
 >>> msg.body = 'This is the plain text body'
 >>> msg.html = 'This is the <b>HTML</b> body'
 >>> with app.app_context():
 ...  mail.send(msg)
 ...
```

Integrating Emails with the Application

Example 6-3. hello.py: email support

```
from flask_mail import Message

app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin
<flasky@example.com>'

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] +
                  subject,  sender=app.config['FLASKY_MAIL_SENDER'],
                  recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

Example 6-4. hello.py: email example

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
  form = NameForm()
  if form.validate_on_submit():
    user = User.query.filter_by(username=form.name.data).first()
    if user is None:
      user = User(username=form.name.data)  db.session.add(user)
      session['known'] = False
      if app.config['FLASKY_ADMIN']:
        send_email(app.config['FLASKY_ADMIN'], 'New User',
            'mail/new_user', user=user)
    else:
      session['known'] = True  session['name'] =
    form.name.data  form.name.data = ''
    return redirect(url_for('index'))
  return render_template('index.html', form=form, name=session.get('name'),
          known=session.get('known', False))
```

For Linux and macOS users, the command to set this variable is:

```
(venv) $ export FLASKY_ADMIN=<your-email-address>
```

For Microsoft Windows users, this is the equivalent command:

```
(venv) $ set FLASKY_ADMIN=<your-email-address>
```

With these environment variables set, you can test the application and receive an email every time you enter a new name in the form.

Sending Asynchronous Email

Example 6-5. hello.py: asynchronous email support

```
from threading import Thread

        def send_async_email(app, msg):  with
          app.app_context():  mail.send(msg)

def send_email(to, subject, template, **kwargs):
  msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
        sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
  msg.body = render_template(template + '.txt', **kwargs)  msg.html =
  render_template(template + '.html', **kwargs)  thr =
  Thread(target=send_async_email, args=[app, msg])  thr.start()
  return thr
```

Large Application Structure

Although having small web applications stored in a single script file can be very con- venient, this approach does not scale well. As the application grows in complexity, working with a single large source file becomes problematic.

Unlike most other web frameworks, Flask does not impose a specific organization for large projects; the way to structure the application is left entirely to the developer. In this chapter, a possible way to organize a large application in packages and modules is presented. This structure will be used in the remaining examples of the book.

Project Structure

Example 7-1 shows the basic layout for a Flask application.

Example 7-1. Basic multiple-file Flask application structure

```
|-flasky
  |-app/
    |-templates/
    |-static/
    |-main/
      |-__init__.py
      |-errors.py
      |-forms.py
      |-views.py
    |-__init__.py
    |-email.py
    |-models.py
  |-migrations/
  |-tests/
    |-__init__.py
    |-test*.py
  |-venv/
  |-requirements.txt
  |-config.py
  |-flasky.py
```

Configuration Options

Example 7-2. config.py: application configuration

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
  SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess
  string'  MAIL_SERVER = os.environ.get('MAIL_SERVER',
  'smtp.googlemail.com')  MAIL_PORT =
  int(os.environ.get('MAIL_PORT', '587'))
  MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() in
    \  ['true', 'on', '1']
  MAIL_USERNAME        =        os.environ.get('MAIL_USERNAME')
  MAIL_PASSWORD        =        os.environ.get('MAIL_PASSWORD')
  FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
  FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'
  FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')
  SQLALCHEMY_TRACK_MODIFICATIONS = False
```

```
  @staticmethod
  def init_app(app):
    pass

class DevelopmentConfig(Config): DEBUG = True
  SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
    'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config): TESTING = True
  SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
    'sqlite://'

class ProductionConfig(Config):
  SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
    'sqlite:///' + os.path.join(basedir, 'data.sqlite')

config = {
  'development': DevelopmentConfig, 'testing':
  TestingConfig, 'production': ProductionConfig,

  'default': DevelopmentConfig
}
```

Example 7-3. app/_init_.py: application package constructor

```
from flask import Flask, render_template  from
flask_bootstrap import Bootstrap  from flask_mail
import Mail
from flask_moment import Moment
from flask_sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()  mail = Mail()
moment = Moment()  db = SQLAlchemy()

def create_app(config_name):  app = Flask(__name__)
  app.config.from_object(config[config_name])
  config[config_name].init_app(app)

  bootstrap.init_app(app)  mail.init_app(app)
  moment.init_app(app)  db.init_app(app)

  # attach routes and custom error pages here

  return app
```

Implementing Application Functionality in a Blueprint

Example 7-4. app/main/_init_.py: main blueprint creation

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

Example 7-5. app/_init_.py: main blueprint registration

```
def create_app(config_name):
  # ...

  from .main import main as main_blueprint
  app.register_blueprint(main_blueprint)

  return app
```

Example 7-6. app/main/errors.py: error handlers in main blueprint

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def page_not_found(e):
 return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
 return render_template('500.html'), 500
```

A difference when writing error handlers inside a blueprint is that if the `errorhandler` decorator is used, the handler will be invoked only for errors that orig- inate in the routes defined by the blueprint.

Example 7-7. app/main/views.py: application routes in main blueprint

```
from datetime import datetime
from flask import render_template, session, redirect, url_for
from . import main
from .forms import NameForm
from .. import db
from ..models import User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('.index'))
    return render_template('index.html',
                           form=form, name=session.get('name'),
                           known=session.get('known', False),
                           current_time=datetime.utcnow())
```

Application Script

The *flasky.py* module in the top-level directory is where the application instance is defined. This script is shown in Example 7-8.

Example 7-8. flasky.py: main script

```
import os
from app import create_app, db  from app.models import User,
Role  from flask_migrate import Migrate

app = create_app(os.getenv('FLASK_CONFIG') or 'default')  migrate =
Migrate(app, db)

@app.shell_context_processor
def make_shell_context():
  return dict(db=db, User=User, Role=Role)
```

For Linux and macOS, this is all done as follows:

```
(venv) $ export FLASK_APP=flasky.py
(venv) $ export FLASK_DEBUG=1
```

And for Microsoft Windows:

```
(venv) $ set FLASK_APP=flasky.py
(venv) $ set FLASK_DEBUG=1
```

Requirements File

This file can be generated automatically by *pip* with the following command:

```
(venv) $ pip freeze >requirements.txt
```

It is a good idea to refresh this file whenever a package is installed or upgraded. An example requirements file is shown here:

```
alembic==0.9.3 blinker==1.4 click==6.7
dominate==2.3.1 Flask==0.12.2
Flask-Bootstrap==3.3.7.1 Flask-Mail==0.9.1
Flask-Migrate==2.0.4 Flask-Moment==0.5.1 Flask-
SQLAlchemy==2.2 Flask-WTF==0.14.2
itsdangerous==0.24 Jinja2==2.9.6 Mako==1.0.7
MarkupSafe==1.0
python-dateutil==2.6.1 python-editor==1.0.3 six==1.10.0
SQLAlchemy==1.1.11 visitor==0.1.3 Werkzeug==0.12.2
WTForms==2.1
```

When you need to build a perfect replica of the virtual environment, you can create a new virtual environment and run the following command on it:

```
(venv) $ pip install -r requirements.txt
```

The version numbers in the example *requirements.txt* file are likely going to be outda- ted by the time you read this. You can try using more recent releases of the packages, if you like.

Unit Tests

This application is very small, so there isn't a lot to test yet. But as an example, two simple tests can be defined, as shown in Example 7-9.

Example 7-9. tests/test_basics.py: unit tests

```
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):  def setUp(self):
    self.app = create_app('testing')  self.app_context =
    self.app.app_context()  self.app_context.push()
    db.create_all()
  def tearDown(self):  db.session.remove()  db.drop_all()
    self.app_context.pop()

  def test_app_exists(self):  self.assertFalse(current_app is None)

  def test_app_is_testing(self):
    self.assertTrue(current_app.config['TESTING'])
```

The tests are written using the standard `unittest` package from the Python standard library. The `setUp()` and `tearDown()` methods of the test case class run before and after each test, and any methods that have a name that begins with `test_` are executed as tests.

> If you want to learn more about writing unit tests with Python's `unittest` package, read the official documentation.

Example 7-10. flasky.py: unit test launcher command

```
@app.cli.command()
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

The implementation of the `test()` function invokes the test runner from the `unittest` package.

The unit tests can be executed as follows:

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok

.----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
----
```

Database Setup

Regardless of the source of the database URL, the database tables must be created for the new database. When working with Flask-Migrate to keep track of migrations, database tables can be created or upgraded to the latest revision with a single com- mand:

```
(venv) $ flask db upgrade
```

Running the Application

The refactoring is now complete, and the application can be started. Make sure you have updated the FLASK_APP environment variable as indicated in "Application Script" on page 93, and then run the application as usual:

```
(venv) $ flask run
```

Having to set the FLASK_APP and FLASK_DEBUG environment variables every time a new command-prompt session is started can get tedious, so you should configure your system so that these variables are set by default. If you are using *bash*, you can add them to your ~/*.bashrc* file.

Believe it or not, you have reached the end of Part I. You have now learned about the basic elements necessary to build a web application with Flask, but you probably feel unsure about how all these pieces fit together to form a real application. The goal of Part II is to help with that by walking you through the development of a complete application.