Chapter 18: Integrating with Legacy Databases and Applications

Django is best suited for so-called green-field development – that is, starting projects from scratch, as if you were constructing a building on a fresh field of green grass. But despite the fact that Django favors from-scratch projects, it's possible to integrate the framework into legacy databases and applications. This chapter explains a few integration strategies.

Integrating with a Legacy Database

Django's database layer generates SQL schemas from Python code – but with a legacy database, you already have the SQL schemas. In such a case, you'll need to create models for your existing database tables. For this purpose, Django comes with a tool that can generate model code by reading your database table layouts. This tool is called `inspectdb`, and you can call it by executing the command`manage.py inspectdb`.

Using `inspectdb`

The `inspectdb` utility introspects the database pointed to by your settings file, determines a Django model representation for each of your tables, and prints the Python model code to standard output.

Here's a walk-through of a typical legacy database integration process from scratch. The only assumptions are that Django is installed and that you have a legacy database.

1. Create a Django project by running `django-admin.py startproject mysite` (where `mysite` is your project's name). We'll use `mysite` as the project name in this example.

2. Edit the settings file in that project, `mysite/settings.py`, to tell Django what your database connection parameters are and what the name of the database is. Specifically, provide the `DATABASE_NAME`,`DATABASE_ENGINE`, `DATABASE_USER`, `DATABASE_PASSWORD`, `DATABASE_HOST`, and `DATABASE_PORT` settings. (Note that some of these settings are optional. Refer to Chapter 5 for more information.)

3. Create a Django application within your project by running `python mysite/manage.py startapp myapp`(where `myapp` is your application's name). We'll use `myapp` as the application name here.

4. Run the command `python mysite/manage.py inspectdb`. This will examine the tables in the`DATABASE_NAME` database and print the generated model class for each table. Take a look at the output to get an idea of what `inspectdb` can do.

5. Save the output to the `models.py` file within your application by using standard shell output redirection:

```
python mysite/manage.py inspectdb >
mysite/myapp/models.py
```

6. Edit the `mysite/myapp/models.py` file to clean up the generated models and make any necessary customizations. We'll give some hints for this in the next section.

Cleaning Up Generated Models

As you might expect, the database introspection isn't perfect, and you'll need to do some light cleanup of the resulting model code. Here are a few pointers for dealing with the generated models:

1. Each database table is converted to a model class (i.e., there is a one-to-one mapping between database tables and model classes). This means that you'll need to refactor the models for any many-to-many join tables into `ManyToManyField` objects.

2. Each generated model has an attribute for every field, including `id` primary key fields. However, recall that Django automatically adds an `id` primary key field if a model doesn't have a primary key. Thus, you'll want to remove any lines that look like this:

```
3.id = models.IntegerField(primary_key=True)
```

Not only are these lines redundant, but also they can cause problems if your application will be adding*new* records to these tables.

4. Each field's type (e.g., `CharField`, `DateField`) is determined by looking at the database column type (e.g., `VARCHAR`, `DATE`). If `inspectdb` cannot map a column's type to a model field type, it will use`TextField` and will insert the Python comment `'This field type is a guess.'` next to the field in the generated model. Keep an eye out for that, and change the field type accordingly if needed.

   If a field in your database has no good Django equivalent, you can safely leave it off. The Django model layer is not required to include every field in your table(s).

5. If a database column name is a Python reserved word (such as `pass`, `class`, or `for`), `inspectdb` will append `'_field'` to the attribute name and set the `db_column` attribute to the real field name (e.g.,`pass`, `class`, or `for`).

   For example, if a table has an `INT` column called `for`, the generated model will have a field like this:

```
for_field = models.IntegerField(db_column='for')
```

   `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'`next to the field.

6. If your database contains tables that refer to other tables (as most databases do), you might need to rearrange the order of the generated models so that models that refer to other models are ordered properly. For example, if model `Book` has a `ForeignKey` to model `Author`, model `Author` should be defined before model `Book`. If you need to create a relationship on a model that has not yet been defined, you can use a string containing the name of the model, rather than the model object itself.

7. `inspectdb` detects primary keys for PostgreSQL, MySQL, and SQLite. That is, it inserts`primary_key=True` where appropriate. For other databases, you'll need to insert `primary_key=True` for at least one field in each model, because Django models are required to have a `primary_key=True` field.

8. Foreign-key detection only works with PostgreSQL and with certain types of MySQL tables. In other cases, foreign-key fields will be generated as`IntegerField``s, assuming the foreign-key column was an ``INT` column.

Integrating with an Authentication System

It's possible to integrate Django with an existing authentication system – another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It would be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

To handle situations like this, the Django authentication system lets you plug in other authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

Specifying Authentication Backends

Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication. When somebody calls `django.contrib.auth.authenticate()` (as described in Chapter 14), Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the `AUTHENTICATION_BACKENDS` setting. This should be a tuple of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, `AUTHENTICATION_BACKENDS` is set to the following:

```
('django.contrib.auth.backends.ModelBackend',)
```

That's the basic authentication scheme that checks the Django users database.

The order of `AUTHENTICATION_BACKENDS` matters, so if the same username and password are valid in multiple backends, Django will stop processing at the first positive match.

Writing an Authentication Backend

An authentication backend is a class that implements two methods: `get_user(id)` and`authenticate(**credentials)`.

The `get user` method takes an `id` – which could be a username, database ID, or whatever – and returns a`User` object.

The `authenticate` method takes credentials as keyword arguments. Most of the time it looks like this:

```
class MyBackend(object):

    def authenticate(self, username=None,
password=None):

        # Check the username/password and return a
User.
```

But it could also authenticate a token, like so:

```
class MyBackend(object):

    def authenticate(self, token=None):

        # Check the token and return a User.
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are valid. If they're not valid, it should return `None`.

The Django admin system is tightly coupled to Django's own database-backed `User` object described in Chapter 14. The best way to deal with this is to create a Django `User` object for each user that exists for your backend (e.g., in your LDAP directory, your external SQL database, etc.). Either you can write a script to do this in advance or your `authenticate` method can do it the first time a user logs in.

Here's an example backend that authenticates against a username and password variable defined in your`settings.py` file and creates a Django `User` object the first time a user authenticates:

```
from django.conf import settings

from django.contrib.auth.models import User,
check_password


class SettingsBackend(object):

    """

    Authenticate against the settings ADMIN_LOGIN and
ADMIN_PASSWORD.
```

```
    Use the login name, and a hash of the password. For
example:



    ADMIN_LOGIN = 'admin'

    ADMIN_PASSWORD =
'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'

    """

    def authenticate(self, username=None,
password=None):

        login_valid = (settings.ADMIN_LOGIN ==
username)

        pwd_valid = check_password(password,
settings.ADMIN_PASSWORD)

        if login_valid and pwd_valid:

            try:

                user =
User.objects.get(username=username)

            except User.DoesNotExist:

                # Create a new user. Note that we can
set password

                # to anything, because it won't be
checked; the password

                # from settings.py will.

                user = User(username=username,
password='get from settings.py')

                user.is_staff = True

                user.is_superuser = True

                user.save()

            return user
```

```
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

For more on authentication backends, see the official Django documentation.

Integrating with Legacy Web Applications

It's possible to run a Django application on the same Web server as an application powered by another technology. The most straightforward way of doing this is to use Apache's configuration file, `httpd.conf`, to delegate different URL patterns to different technologies. (Note that Chapter 12 covers Django deployment on Apache/mod_python, so it might be worth reading that chapter first before attempting this integration.)

The key is that Django will be activated for a particular URL pattern only if your `httpd.conf` file says so. The default deployment explained in Chapter 12 assumes you want Django to power every page on a particular domain:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

Here, the `<Location "/">` line means "handle every URL, starting at the root," with Django.

It's perfectly fine to limit this `<Location>` directive to a certain directory tree. For example, say you have a legacy PHP application that powers most pages on a domain and you want to install a Django admin site at `/admin/` without disrupting the PHP code. To do this, just set the `<Location>` directive to `/admin/`:

```
<Location "/admin/">
    SetHandler python-program
```

```
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

With this in place, only the URLs that start with `/admin/` will activate Django. Any other page will use whatever infrastructure already existed.

Note that attaching Django to a qualified URL (such as `/admin/` in this section's example) does not affect the Django URL parsing. Django works with the absolute URL (e.g., `/admin/people/person/add/`), not a "stripped" version of the URL (e.g., `/people/person/add/`). This means that your root URLconf should include the leading `/admin/`.