First Experiments with Django Creating App Database

Maxim Mozgovoy

Creating Our First Project

- Remember, Django is the whole framework, we will use its tools and add code inside ready modules.
- Many operations with Django are performed with `django-admin` tool.
- Let's create the project `FirstPrj`:
  `django-admin startproject FirstPrj`
- This command will create a new project inside the subfolder `FirstPrj` of your current folder.

Running the Project

- Django includes a built-in web server. Let's run the project.
- Go inside `FirstPrj` and execute:
  `python manage.py runserver`
- By default, the server will run on the port 8000.
  We can test it by opening a page
  http://localhost:8000
  in a browser.
- Note: Django server is not intended to be used in real production environment. It is for development only.

Project Structure

Django creates the following files and folders:

```
FirstPrj/
    db.sqlite3          -- default database
    manage.py           -- management tool
    FirstPrj/
        settings.py     -- project configuration
        urls.py         -- URL dispatcher config
        wsgi.py         -- entry-point for a
                           production web server
```

Projects and Apps

- In Django, each *project* can include several *apps*.
- Typically, and *app* is an independent module that can exist apart from the project (and even be included into another project).
- In our course, we will deal with single-app projects only.
- You can create new app `FirstApp` inside the project with `manage.py` tool:

```
python manage.py startapp FirstApp
```

- The code for this app will be stored inside `FirstApp` folder.

Project Structure (with an App)

Django creates the following files and folders:

```
FirstPrj/
    db.sqlite3          -- default database
    manage.py           -- management tool
    FirstApp/
        admin.py        -- admin configuration
        apps.py         -- app configuration
        models.py       -- database schema config
        views.py        -- application layer
    FirstPrj/
        settings.py     -- project configuration
        urls.py         -- URL dispatcher config
        wsgi.py         -- entry-point for a
                           production web server
```


Database Support

- By default, Django assumes that each project needs database access.
- By default, Django creates a separate SQLite database for each new project (it is stored in `db.sqlite3`).
- SQLite is a great simple engine for testing the project and for small applications (the database is stored in a single file and easily accessed from Python)
- For real production environment a larger engine might be necessary (MySQL, PostgreSQL, etc.)
- To connect another engine, one needs to edit `settings.py`

Migrations

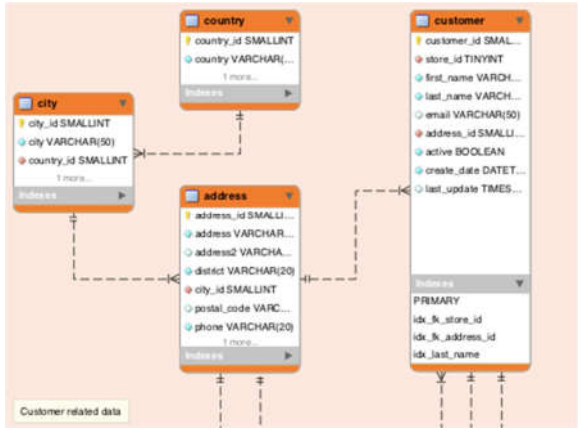
- Inside `settings.py`, there is a section `INSTALLED_APPS`.
- By default it contains the list `'django.contrib.admin'`, `'django.contrib.auth'`...
- These are *default apps*, installed for each new project. Not every project needs all of them, so we can edit this list.
- (We will also have to add our own app to this list later).
- Now the project database is empty. To create necessary data for the installed apps, we need to apply *migrations* for them:
  `python manage.py migrate`
- Every time we install or modify an app and need to change the database schema, we have to create and apply new migrations:
  `python manage.py makemigrations <app-name>`
  `python manage.py migrate`

Administering the Website

- Django includes a build-in website administration app.
- In particular, you can see your database schema there.
- To run it, we first need to create a *superuser* who will have full access to the project data:
  `python manage.py createsuperuser`
- (Input any username, email and password)
- To access admin panel, just go to the `/admin` subfolder of the website (e.g., `http://localhost:8000/admin`)

Create Database Schema

- Normally, programmers design database schemes using specialized “data modelers” or plain SQL.
- In Django, it is done with specialized Python data types.
- All tables should be declared in the file `models.py` inside the app folder.



Creating Entities

Each entity (a table) should be declared in a subclass of models.Model. Class members will be database columns:

```
class Person(models.Model):
    name = models.CharField(max_length=20)
    height = models.IntegerField()
    date_of_birth = models.DateField()
```

id	name	height	date_of_birth
1	John	180	10.10.1990
2	Mike	175	20.03.1984
3	Mary	174	07.11.1970

Note: by default in each table Django creates a column “**id**”. It contains a unique integer identifier of a record.

Common Field Types

- `CharField(max_length)` -- fixed-length strings
- `DateField` -- date type
- `TimeField` -- time type
- `DateTimeField` -- date & time type
- `EmailField` -- email
- `FileField` -- file storage
- `FloatField` -- real numbers
- `ImageField` -- images
- `IntegerField` -- integers
- `TextField` -- variable-length large string of text

Foreign Keys (Many to One Rels)

To refer a value in another table, use foreign keys:

```
class Manufacturer(models.Model):
    name = models.CharField(max_length=20)

class Car(models.Model):
    number_plate = models.CharField(max_length=10)
    owner = models.CharField(max_length=20)
    manufacturer = models.ForeignKey(Manufacturer,
                        on_delete=models.CASCADE,
                        related_name='cars')
    checkup_date = models.DateField()
```

Behavior of `on_delete`:

CASCADE: delete references to the object to be deleted

PROTECT: prevent deletion of referenced keys

Many to Many Relationships

Django has built-in methods for many-to-many relationships.

Traditional approach:

```
class Person(models.Model): # Mike, John, Mary
    name = models.CharField(max_length=20)

class Group(models.Model): # Students, Golfers
    name = models.CharField(max_length=20)

class PersonGroupRels(models.Model):
    person = models.ForeignKey(Person,
                        on_delete=models.CASCADE)
    group = models.ForeignKey(Group,
                        on_delete=models.CASCADE)
```


Many to Many Relationships

Django `ManyToManyField`:

```
class Group(models.Model): # Students, Golfers
    name = models.CharField(max_length=20)

class Person(models.Model): # Mike, John, Mary
    name = models.CharField(max_length=20)
    groups = models.ManyToManyField(Group)
```

(The intermediate relation will be generated automatically)

Final Touches

Let's add to FirstApp (`models.py` file) the following entities:

```
class Manufacturer(models.Model):
    name = models.CharField(max_length=20)
    year_founded = models.IntegerField()

class Car(models.Model):
    number_plate = models.CharField(max_length=10)
    owner = models.CharField(max_length=20)
    manufacturer = models.ForeignKey(Manufacturer,
                        on_delete=models.CASCADE)
    checkup_date = models.DateField()
```

Final Touches

- Once again, let's review the project:

```
FirstPrj\
    FirstPrj\settings.py
    FirstApp\models.py
```

- We need to add `FirstApp` into `INSTALLED_APPS` inside `settings.py`:

```
INSTALLED_APPS = ['FirstApp',...]
```

- Next, let's create and apply migrations:

```
python manage.py makemigrations FirstApp
python manage.py migrate
```


Final Touches

- At this point, the database is modified. Now we need to make it visible in the administration console.
- Open `FirstApp\admin.py`, then import and register there each class you created in `models.py`:

```
from FirstApp.models import Manufacturer, Car
admin.site.register(Manufacturer)
admin.site.register(Car)
```

- Open the administration console and check new tables (http://localhost:8000/admin).
- Note that in most cases you don't need to restart the server after making changes!

Final Touches

- When adding data, you will notice that each table element is shown as an automatic name such as "*Manufacturer object (1)*".
- To make names more readable, you can provide a custom textual label by providing `__str__()` function:

```
class Manufacturer(models.Model):
    name = models.CharField(max_length=20)
    year_founded = models.IntegerField()

    def __str__(self):
        return self.name
```


Coming Exercise

During the next exercise session you have to implement the database schema for your application.

All the tables should be filled with reasonable data and manageable via the standard Django admin interface.