

Implementing the Backend Data Model and Persistence Layer with Django Models

In this chapter, we will see how to implement the backend **data model** and **persistence layer** and learn how to install and setup the application from the project **Git** repository. You will need a set of tools installed on your computer as prerequisites and you can install them by following the instructions contained in the **README.md** file in the main folder of the Git repository. Once the environment is installed you will learn how to implement the persistence layer with Django.

Every application needs a **domain model** to represent the components of the business. That means something written in the code that resembles the real-world functionality for our business. In the example for this book, we will need some models to represent that as will be explained in more detail later in this chapter.

Also, most applications need to store state in a persistent manner, which means saving changes to information and state permanently, usually in a database. Persistence is essential to avoid losing information after an application restarts or fails. For this to happen we need to implement a persistence layer, which will be responsible for storing the information in our application.

Django provides its own **ORM** framework for persistence out-of-the-box, that will allow a developer to automatically generate this magic with minimal code as you will discover later in the chapter.

In this chapter we are going to cover the following main topics:

- Installing the development environment
- Installing and configuring the project
- Implementing the persistence layer with Models

Technical requirements

The `README.md` is the most important file in the project as it is the starting point for any new developer joining it and should contain precious information like the installation instructions for the application as well as the project's best practices used in coding and management of the software lifecycle. Should contain information like naming conventions in source code, naming conventions in **branches and commits**, **merge requests (MR)**, **code reviews**, **Definition of Done (DoD)** for tasks, etc. To avoid the file becoming too big, it can contain links to further information in the company's content management sites like Atlassian Confluence. The README file is written using the Markdown syntax, therefore the file extension `.md`; this syntax is commonly used for documentation in content management tools.

You should be able to install the application that contains the source code used in this book by following the instructions in the `README.md` file without any inconsistency or ambiguity. It should be clear to understand and follow if it is rigorous and is up to date.

If you join a project where the README file is empty or missing, that speaks very poorly about the project itself and most certainly means that the quality of the application is going to be similar, that is, extremely poor. This file is the welcome introduction for someone joining a project so is the first impression they will get of it.

Specific hardware/software requirements

Hardware

- Minimum memory: 4 GB

Software

- Git
- Python 3
- Django 5

This project has been created with Python version 3.10.12 and Django version 5.0.4 on a machine with Ubuntu OS version 22.04.1

Installing the development environment

Before starting to build our backend application we will need a set of tools installed on your computer as prerequisites (see [Technical requirements](#) section). We can install them by following the instructions contained in the `README.md` file in the backend main folder of the Git repository (<https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/blob/main/backend/README.md>).

I have installed the environment on a machine with Ubuntu OS version 22.04.1, but if you have a different operating system on your computer, you can find links for installation in different OSes in the README file.

To proceed with the following installation steps, you need to open a terminal in your computer and execute the required commands that are explained as follows.

Git installation

First you will need to install Git to be able to clone the Git repository for the application.. Execute the following commands:

```
$ sudo apt install git-all
$ git -version
```

If Git is installed, you should see the version of your installation:

```
git version 2.34.1
```

For other operating systems, see the following link:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Creating the Python environment

It is a good practice to create a Python environment for our backend application to avoid conflicts with libraries used by other projects. This way we can be sure that the libraries used by our project are the only ones that we have installed on purpose for it in

its own isolated environment, so it will never use other libraries that are used by other projects on your computer. This way we can avoid problems produced by different versions of these libraries that may have been installed on your computer before.

Inside the folder where later you will clone the source code for this project, run the following commands for doing the following:

- Installing `virtualenv`:

```
$ python -m pip --version
$ python -m pip install virtualenv
```

- Creating a virtual environment:

```
$ python -m venv packtDjango-venv
```

- Activating/using the virtual environment:

```
$ source packtDjango-venv/bin/activate (Linux)
$ packtDjango-venv\Scripts\activate (Windows)
```

Python installation

Next you will need to install Python version 3 to be able to work with the Django framework. Execute the following commands:

```
$ sudo apt update
$ sudo apt install python3
$ python3 --version
```

If Python is installed, you should see the version of your installation:

```
Python 3.10.12
```

For other operating systems see the following link:

<https://kinsta.com/knowledgebase/install-python/>

Django installation

Next you will need to install Django version 5 to be able to implement the application. Execute the following commands:

```
$ python -m pip install Django
$ python -m django --version
$ python -m pip install django-environ
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

For other operating systems see the following link:

<https://docs.djangoproject.com/en/5.0/topics/install/>

After installing all the tools, you can see the libraries used in the project with the following command:

```
$ pip freeze
asgiref==3.8.1
beautifulsoup4==4.12.3
behave==1.2.6
behave-django==1.4.0
Django==5.0.6
parameterized==0.9.0
parse==1.20.1
parse-type==0.6.2
psycpg2-binary==2.9.9
six==1.16.0
soupsieve==2.5
sqlparse==0.5.0
typing_extensions==4.11.0
```

The libraries highlighted are used for testing Django apps, and as you can see in the `README.md` file, they are installed separately because they are not included in the Django distribution out-of-the-box. But we will talk about testing with more details in the last chapters of the book.

Once you have installed all the tools required on your computer by the backend application, you can proceed to the next step, which consists of installing the example backend application for this tutorial on your computer.

Installing and configuring the project

Now that you have the required tools installed in your development environment you can go ahead and clone the git repository that holds the source code of this book in a local folder of your choice on your computer. You can use the git clone command in a terminal or with your preferred IDE. I recommend using Visual Studio Code IDE because it has good support for both Django and Angular:

```
$ git clone https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular.git
```

Once you have cloned the git repository in your computer, you can find the `README.md` file in the main folder of the project where you can see the same installation instructions as in this book.

Now you have all the necessary tools and source code to explore and play with the provided code, to run it and execute the automated tests.

You can also find an optional section in the README file where you can see the steps to follow when initializing the Django application from scratch instead of downloading the provided application.

In case you'd like to start your own project, run the following command:

```
$ django-admin startproject packtDjangoProject
```

Remember to replace the parameter `packtDjangoProject` with your own choice for your project name.

Projects and applications

Projects and applications are different concepts. In Django, a project is a top-level container for your web application. A project contains the configurations for your web application and one or more apps, each providing a specific set of functionalities or features for your web application. It also contains any components shared between its different apps.

You will also need to create an application to add it to your project to make Django work.

Here's how to create your own application:

Make sure you are in the same directory as `manage.py` and run the following command:

```
$ python manage.py startapp packtDjangoApp
```

Remember to replace the parameter `packtDjangoApp` with your own choice for your application name.

manage.py

The `manage.py` file is a command-line utility that lets you interact with this Django project in several ways. You will need to run it from Python every time you want to perform the most common operations like starting the server, migrating the database, and running tests. Replace the parameter `packtDjangoApp` by your own choice for your application name.

As mentioned earlier Django has separate concepts of **application** and **project**. In any case after the initialization, you will see a project structure like this:

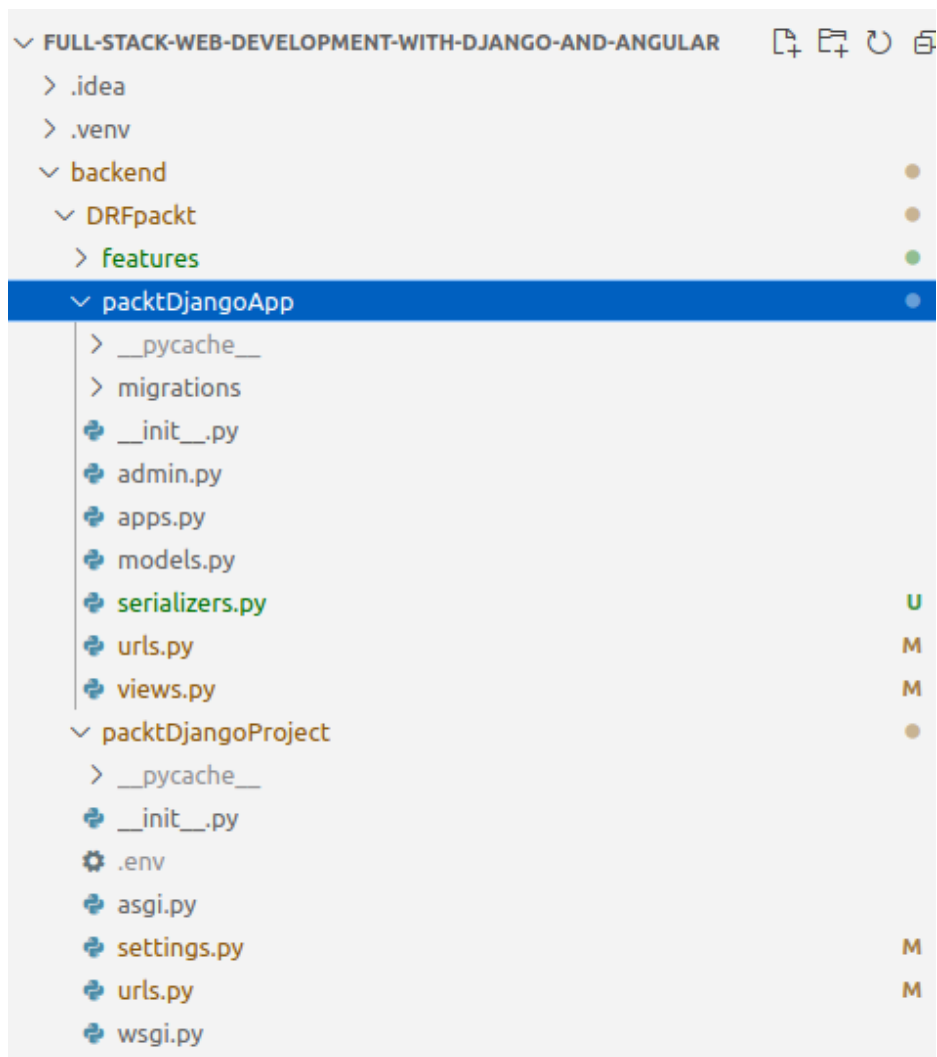


Figure 2.1 – Project initial skeleton and main files

Here is a description of the main files and structure of the generated project:

- The outer `DRFPackt/` root directory: This is a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- `manage.py`: A command-line utility that lets you interact with this Django project in several ways.

- The inner `packtDjangoApp/` directory: This is the actual Python package for your application. Its name is the Python package name you will need to use to import anything inside it (e.g., `packtdjangoApp.urls`).
- The inner `packtDjangoProject/` directory: This is the actual Python package for your project.
- `packtDjangoProject/ __init__.py`: An empty file that tells Python that this directory should be considered a Python package.
- `packtDjangoProject/settings.py`: Settings/configuration for this Django project.
- `packtDjangoApp/urls.py`: The URL declarations for this Django project; a “table of contents” of your Django-powered site.
- `packtDjangoProject /asgi.py`: An entry-point for ASGI-compatible web servers to serve your project.
- `packtDjangoProject /wsgi.py`: An entry-point for WSGI-compatible web servers to serve your project.

We need to add our application to the project. The way to do it is to include it in one configuration file in the project. The `backend/packtDjangoProject/packtDjangoProject/settings.py` file shown in the *Figure 2.1* is one of the most important files in our project because it contains the configuration of the applications installed on it, apart from other important configurations.


By default, the project comes preinstalled with some applications, within the `settings.py` file, you will see the `INSTALLED_APPS` block, which holds the following apps, all of which come preinstalled with Django:

- `django.contrib.admin`: The admin site.
- `django.contrib.auth`: An authentication system.

- `django.contrib.contenttypes`: A framework for content types.
- `django.contrib.sessions`: A session framework.
- `django.contrib.messages`: A messaging framework.
- `django.contrib.staticfiles`: A framework for managing static files.

To include our application in the installed applications for our project, add the following line in the project's `settings.py` file:

```
'packtDjangoApp.apps.PacktdjangoappConfig',
```



```
32
33 INSTALLED_APPS = [
34     'packtDjangoApp.apps.PacktdjangoappConfig',
35     'django.contrib.admin',
36     'django.contrib.auth',
37     'django.contrib.contenttypes',
38     'django.contrib.sessions',
39     'django.contrib.messages',
40     'django.contrib.staticfiles',
41 ]
```

Figure 2.2 – Include application in Django Project

Keep in mind that `PacktdjangoappConfig` is located in the `apps.py` file and matches the name for our application. In the generic case of a different application name, it should be `[Your_App_Name]Config`. This way Django will find the new models and will update the database with new tables or any necessary changes (database migration).

Once we have installed our Django project and included our application in it, we have all the necessary components to proceed with the next step and implement the persistence layer for our application with models.

Implementing the persistence layer with Models

It's now time to implement the layer that communicates with the database, where the application will persist its state information. The persistence layer is always needed to store the information and state of our application permanently, usually in a database.

The persistence layer is the code of the application that connects to the database, so it is usually implemented with the help of a persistence framework also called **ORM** (object relational mapping), that in this case is provided by Django out-of-the-box.

The Django framework is of great aid because it saves most of the development effort needed in dealing with database queries and schema creation/maintenance, helping developers to save precious time just concentrating on coding other topics like the logic of the application.

Maintenance and code quality are also dramatically improved with the help of a persistence framework as you have much fewer lines of code to take care of.

Even though there is less code to maintain, that is not an excuse to not test extensively the persistence layer, as it will avoid many bugs and problems slipping in production. Some developers do not test the persistence layer with the argument that the framework is already tested by the creators. It is not the framework what we need to test but the use we make of it.

As you will find later in **Chapter 8** and in the git repository for this project the persistence layer is extensively tested beforehand. As it is the only way to demonstrate that the models behave as expected.

The Django framework uses a component called **Models** that defines the representation in Python code of our business entities (**ORM** framework).

Django reads this code and generates the database schema for us, saving all the effort generating the **SQL DDL** instructions for us.

Next, we will be generating the database schema that Django needs to run by default based on the preinstalled applications.

Keep in mind that the next steps have already been executed and implemented in the application provided in the Git repository. So, they are just for your reference in case you wish to implement your own application.

Database initialization

As you have already seen in **Figure 2.7**, the `settings.py` file is where the database configuration resides, these are the lines for database configuration contained in this file by default:

```
DATABASES = {
```

```
'default': {  
    'ENGINE': 'django.db.backends.sqlite3',  
    'NAME': BASE_DIR / 'db.sqlite3',  
}  
}
```

By default, Django comes preinstalled with the `sqlite3` database, which is included in Python. When starting your first real project however, you may want to use a more scalable database like PostgreSQL, to avoid database-switching headaches down the road.

You can also see in [Figure 2.7](#) the location for the database file a.k.a. `db.sqlite3`

To switch to Postgres DB for example, just add a new database in `settings.py`:

```
DATABASES = {  
    'sqlite3-packtdjango': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    },  
    'postgres-packtdjango': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': DB_NAME,  
        'USER': DB_USER,  
        'PASSWORD': DB_PASSWORD,  
        'HOST': DB_HOST,  
        'PORT': DB_PORT,  
    }  
}
```

Database configurations contain sensitive information, so it is better to put them in a file containing environment variables to avoid storing them in the source code repository. Storing sensitive information together with the source code in the same repository is highly discouraged because somebody could access it, something we should avoid for security purposes. So is a common practice to store it in a file that is not included in the repository.

I have stored it in the repository for this tutorial to see how it looks like, but never forget to exclude it from your repository in a real project and just keep it locally.

To include environment variables in Django, you need to manually install the `django-environ` library because it is not part of the Django distribution.

You can install it with the following command:

```
$ python -m pip install django-environ
```

You also need to create a file manually to store these variables. The environment variables are stored in a file called `.env` located in the same folder as the `settings.py` file (see [Figure 2.7](#)). Keep in mind that this file must be created manually (See README file).

Next you can install the Postgres database manually or use a `Docker` image.

I provided a docker configuration file in the `docker` folder (see [Figure 2.7](#)).

You need to install and start docker in your machine first, but this is out of the scope of this book. For references on how to install Docker on your machine look at this link: <https://docs.docker.com/engine/install/>

Start the Postgres container with the following command:

```
$ docker-compose -f docker-compose.yml up -d
```

To initialize the database schema (only in case you want to start the application from scratch but not necessary if you clone the book's application), you must execute the following command:

```
$ python manage.py migrate
```

After running the command, it will create new tables in the database, depending on the preinstalled applications:

<code>auth_group</code>	<code>auth_user_user_permissions</code>
<code>auth_group_permissions</code>	<code>django_admin_log</code>
<code>auth_permission</code>	<code>django_content_type</code>
<code>auth_user</code>	<code>django_migrations</code>
<code>auth_user_groups</code>	<code>django_session</code>

Next, we will add the models necessary to generate the changes in the database that will implement the business of our application

Creating models

We are going to implement a simple domain model for our application consisting of four new models that we'll be creating, and reuse the `User` model from the `django.contrib.auth` application installed by default before in the **Database initialization** section when we created the initial database and generated the (Table: `auth_user`). The User model is part of the Django distribution that comes preinstalled with an application that implements authorization based on users and roles. So, we do not need to create a model for Users because Django already did it for us.

The domain model for this example represents the business used in the accounting department of a big online store, where each customer has an associated account and invoices for purchased products.

Following is a graphical representation of the domain model in a **UML** (Unified Modelling Language) diagram, where the associations between models are drawn as arrows with a diamond in one edge.

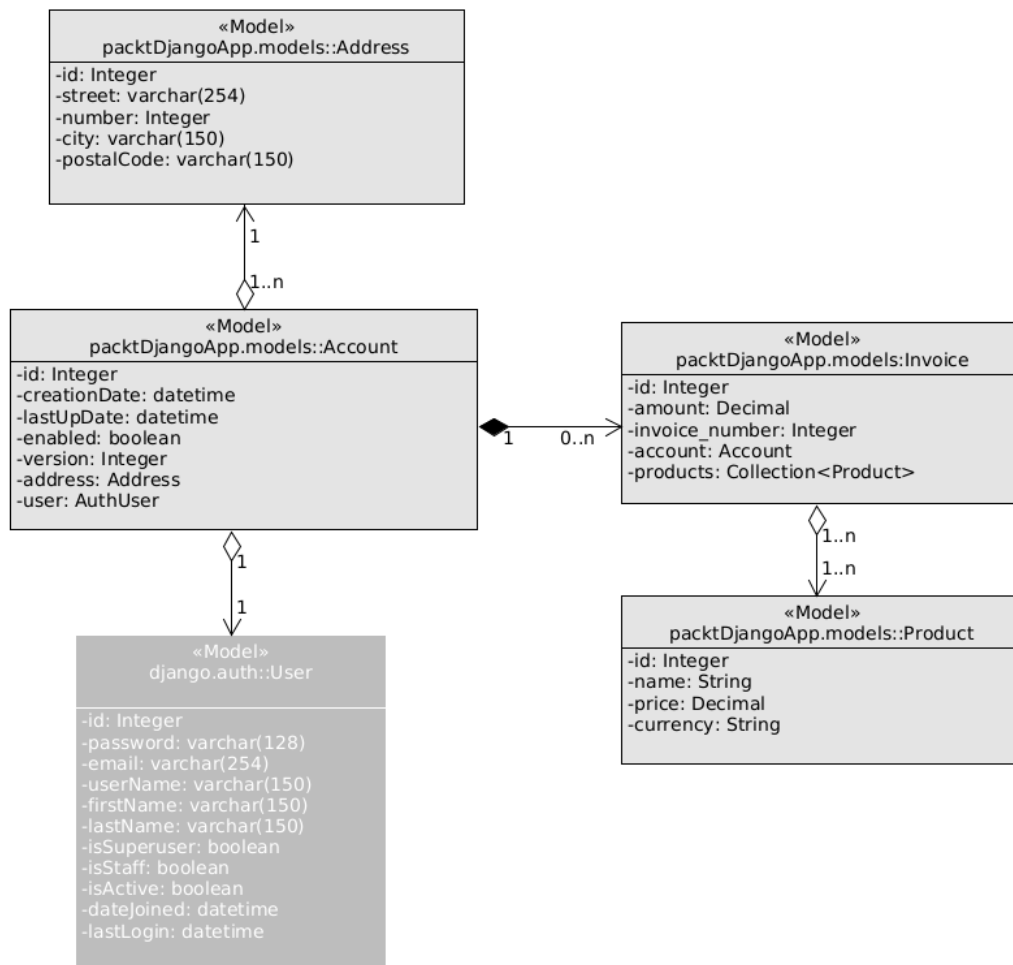


Figure 2.3 – Domain model diagram The models used for our business implementation are shown in light grey, whereas the model that is created by the `django.contrib.auth` application (User) is shown in dark grey.

Let us proceed with the explanation of this domain model:

- User:** This model has already been created by the `django.contrib.auth` application, so we can reuse it to have authenticated users. These users are uniquely identified by the email address.

- **Account:** This model is associated with a unique **User**. That means one **Account** belongs to a single **User** (Aggregation).
- **Address:** This model can be associated with any **Account**. That means one **Address** can belong to any **Account**, but each Account has only one Address.
- **Invoice:** This model is associated with a unique **Account**. That means an **Account** can have any number of invoices and each invoice belongs to a single **Account**.
- **Product:** This model can be associated with any **Invoice**. That means an **Invoice** can have many distinct **Products** and any **Product** can be included in any **Invoice**.

The following diagram shows the database implementation or schema of the domain model where you can see the tables and fields that will be created by Django from the information contained in our new models when we recreate the database later:

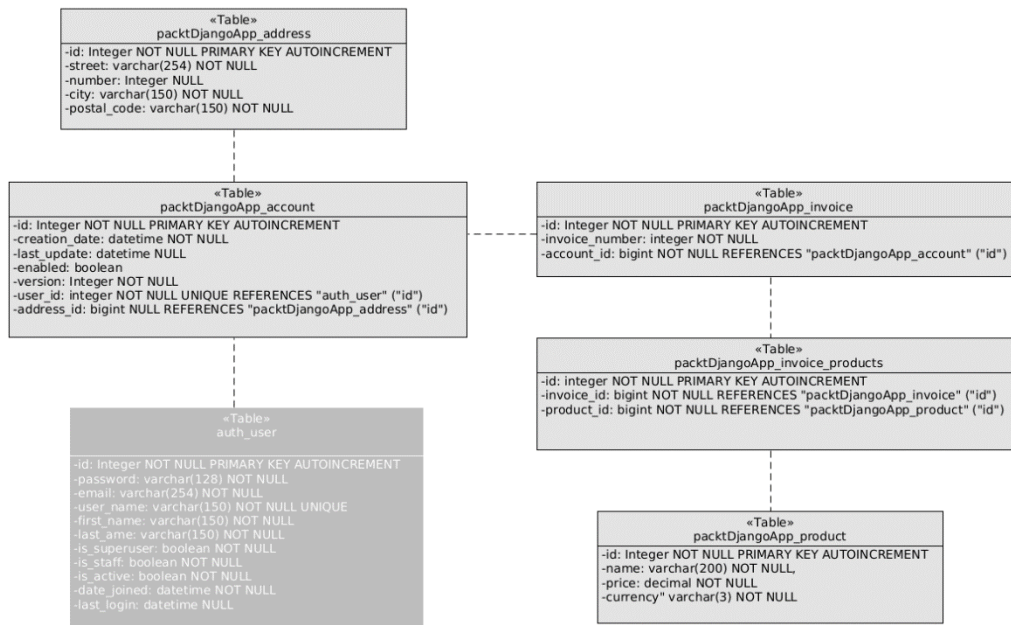


Figure 2.4 – Database schema diagram

The tables that are created from our models are shown in light grey, while the table that is created by the `django.contrib.auth` application (`auth_user`) is shown in dark grey.

There is one table for each model and there is an extra table (`packtDjangoAPP_invoice_products`) needed to implement the association of `Products` with `Invoices`.

Next, we will need to create a superuser (admin) to be able to login into the admin application that comes preinstalled with Django.

Setting up the Admin application (optional)

The `django.contrib.admin` 'Admin' application will allow us to manually manage records in the database. But this application will be explained in more detail later in this section.

To be able to login to the 'Admin' application we need to create a superuser first.

To do so, open a terminal and issue the following command:

```
$ python manage.py createsuperuser
```

You will be prompted to enter a username, an email, a password, and a password confirmation for the superuser. At the end you will get the message `Superuser created successfully`.

Then, register the new models in the `Admin` App.

If you want to be able to manage your models with a user interface (UI) to create, read, update, and delete (CRUD) records in the database you must include them in the `admin.py` file as shown:

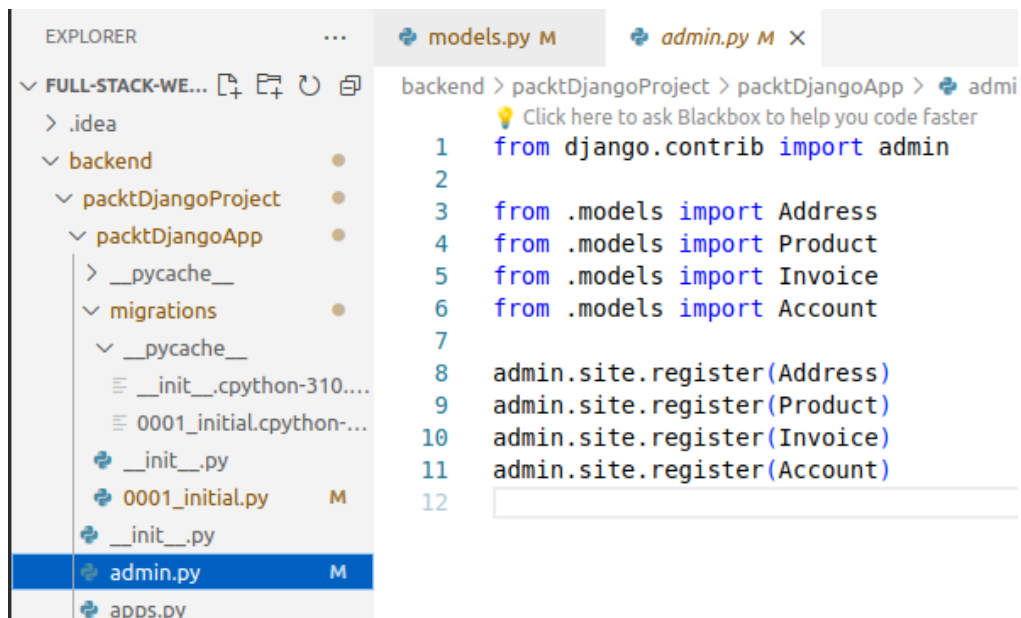


Figure 2.5 - Register models in admin app

Now, log in to the **Admin** app with the superuser created before and manage your model records manually with the UI. You can also manage the security by creating users, groups, and giving access rights. We will look at security in more detail in **Chapter 4**.

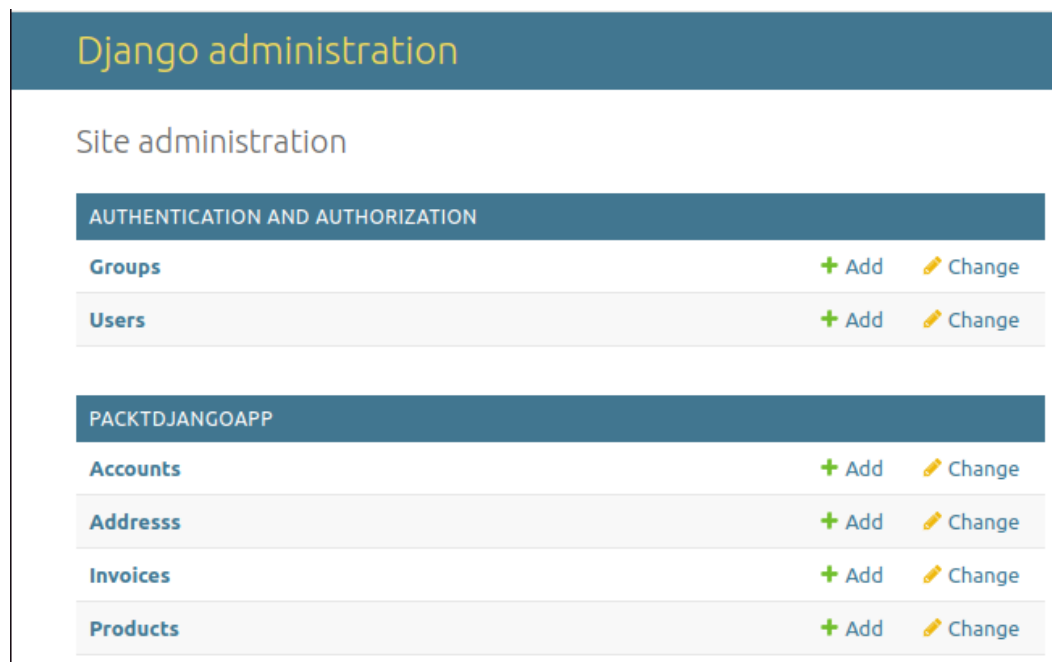


Figure 2.6 - The Admin Application

Next, we will migrate or evolve the database schema, including the new tables that will be created by our new models.

Generating the database schema

Following are the steps to generate the database tables from our models

First, we'll write the code that defines the models

We define the new models in the application in the `models.py` file:

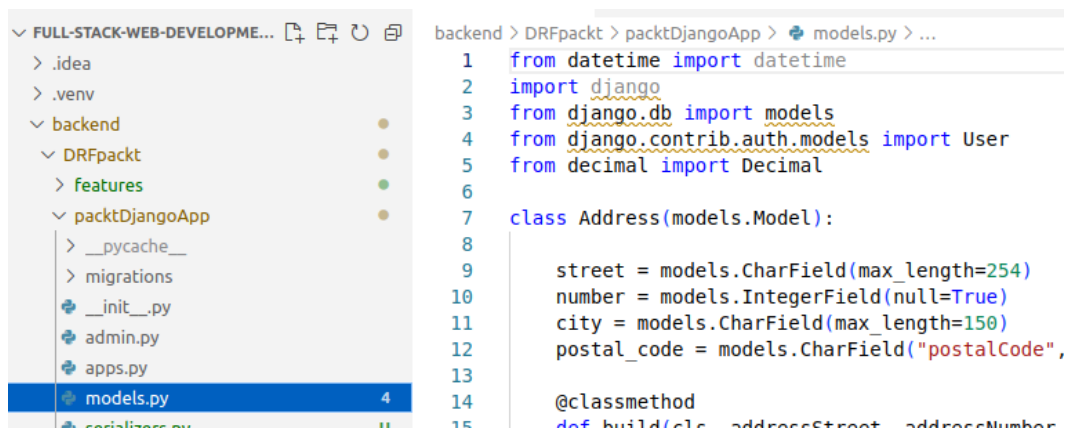


Figure 2.7 - Location of models.py file

Inside this file is where you write the Python code that defines your models:

Let us have a look at the code inside this file and explain some of the main parts of it and how it works:

```
from django.db import models
from django.contrib.auth.models import User

class Invoice(models.Model):

    invoice_number = models.IntegerField(default=0)
    products = models.ManyToManyField(Product)
    account = models.ForeignKey(Account)

    def calculateAmount(self):
        totalAmount = 0
        for product in self.products.all():
            if product.currency == CurrencyEnum.EUR:
```

```
        totalAmount += product.price *  
            Decimal('1.00')
```

```
    if product.currency == CurrencyEnum.USD:  
        totalAmount += product.price *  
            Decimal('0.98')
```

```
    else: totalAmount += 0
```

```
    return totalAmount
```

```
amount = property(calculateAmount)
```

```
class Meta:
```

```
    ordering = ['invoice_number']
```

```
@classmethod
```

```
def build (cls, invoiceNumber):
```

```
    instance = cls(invoice_number=invoiceNumber)
```

```
    return instance
```

```
def __str__(self):
```

```
    return f"Invoice Number: {self.invoice_number}  
    Amount: {self.amount} €"
```

```
def __eq__(self, other):
```

```
    return (
```

```
        isinstance(other, Product) and
```

```
        self.name == other.name
```

```
)
```

```
def __hash__(self):
```

```
return self.pk
```

Let's breakdown the code:

- `from django.db import models`: Imports the Django ORM library that is needed to write models in Python.
- `from django.contrib.auth.models import User`: Imports the `User` model from the `django.contrib.auth` application
- `class Invoice(models.Model)`: Declares a class for a model definition (`Invoice`). Inside this block that is defined by the indentation in Python, the different fields for the model are defined with the required properties (Type, Type of relationship with other models, human-readable name, Max allowed value, empty value is (dis)allowed, default value, etc.)
- `invoice_number = models.IntegerField(default=0)`: One example of field that in this case is declared as `Type` Integer with a default value of `0`
- `def calculateAmount(self)`: Internal method to calculate the amount (one field) based on the prices of the products (another field). This is part of the business logic of our application and is not part of the ORM framework, just for clarification.
- `class Meta`: The field used for ordering in collections `ordering = ['invoice_number']`
- `def build(cls, invoiceNumber)`: A constructor method for instantiation
- `def __str__(self)`: The written representation of the model based on the most important fields
- `def __eq__(self, other)`: Method to check equality between instances

- `def __hash__(self)`: Method to generate hash representation of an instance, that is useful for quickly storing/accessing instances of the model contained in Hash tables.

And so far, we are done with the first model for the Product. Django under the hood will apply its magic and automatically generate a persistence layer for us.

Yes, almost codeless. Here we have demonstrated how Django shines for rapid application development:

Next, to generate the database schema for the new models, we do the following:

To generate or update the database every time you make a change in the models you must execute the following command:

```
$ python manage.py makemigrations packtDjangoApp
```

This command will create a new database migration file with the latest changes in the schema under the `migrations` folder:

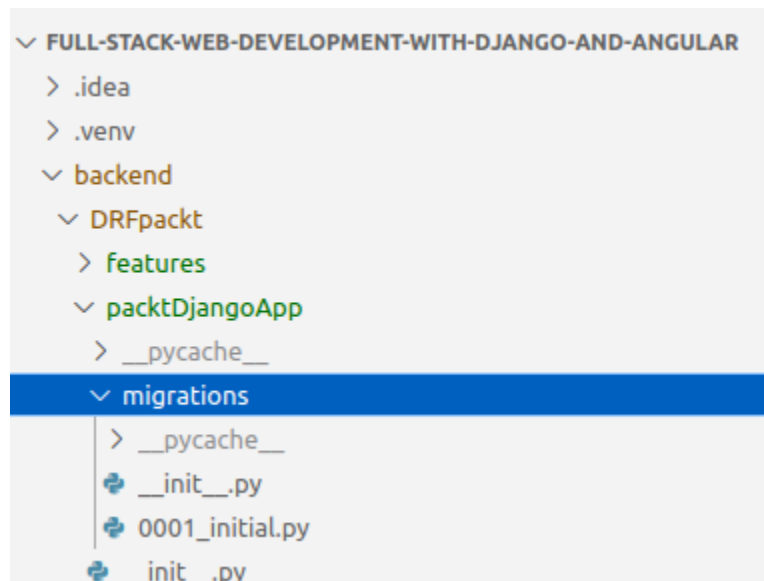


Figure 2.8 – Database migrations folder

As you can see, there is one file that starts with a number prefix and ends with the `.py` suffix (in this case `0001_initial.py` because it is the first change). Django will create one file for every change in the models, so it behaves like a log where you can track and apply the changes in a sequential way. Every new change applied creates a new file incrementing the number prefix of the file:

```
$ python manage.py sqlmigrate packtDjangoApp 0001
```

This command just shows the SQL syntax for the changes declared in the migration file but without applying them in the database. We use it just in case we want to validate the latest changes to the database but without the risk of applying it and effectively modifying the database schema. That means this command is optional to execute the migration.

```
$ python manage.py migrate
```

Applies all the changes effectively in the database and finishes the migration.

Summary

In this chapter, we commenced the practical side of this book.

We started by setting up the development environment with the required tools for coding the backend application. Then, we learned how to set up the application provided for this tutorial or start a new application from scratch. Finally, we learned how to create our own models and reuse the ones provided by Django out-of-the-box and how to generate the database schema with a technique based on migrations.

In the next chapter, we will learn how to expose through REST the backend functionality we created in this chapter with our models.