Django Class Notes

Clarusway

Flight Project 1 - Er diagram, Swagger, Debug Toolbar

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets

Needs

- Python, add the path environment variable
- pip
- virtualenv
- .gitignore file
- .env file

Summary

- Project Introduction
  - Presentation
  - Business Requirements Document (BRD)
  - Entity Relationship (ER) Diagram
- Create project
- Secure your project
  - .gitignore
  - decouple
  - .env
- Publish your project to Github

    - Using VSCode
    - Using git commands
- PostgreSQL setup
- Django Rest Framework setup
- Install Swagger
- Install Debug Toolbar
- Seperate Dev and Prod Settings
    - First Solution: Keeping local settings in "settings_local.py"
    - Second Solution: Separate settings file for each environment
    - Django Settings: Best practices
- Logging

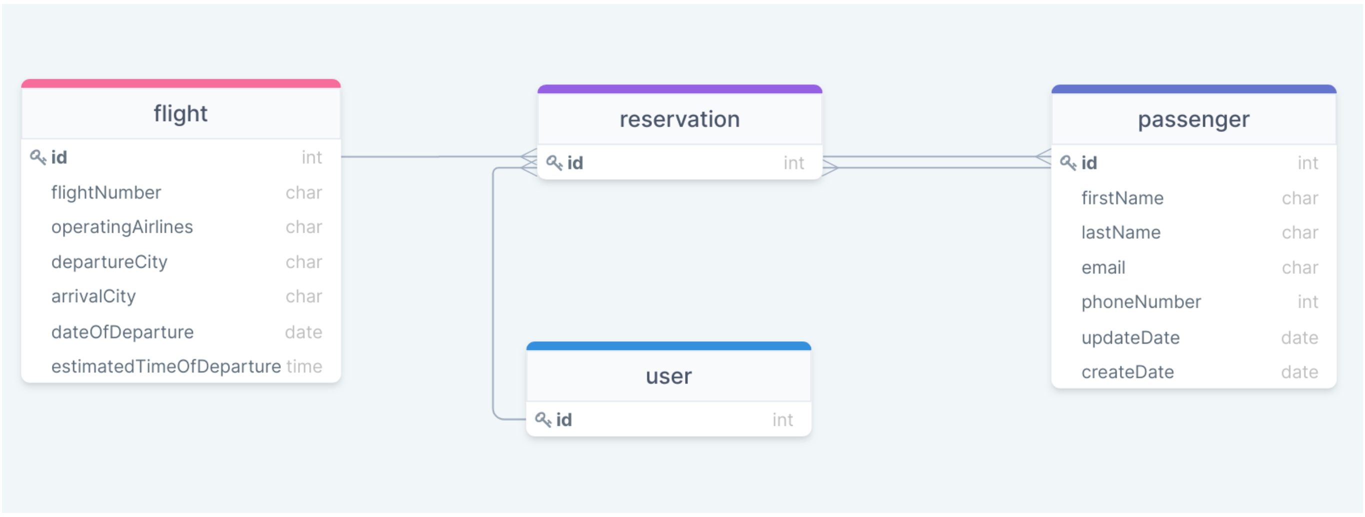
Project Introduction

- Explain project from slides on LMS link

- Explain Business Requirements Document - BRD

- Explain Entity Relationship (ER) Diagram and create the diagram on DrawSQL

- Show a sample finished ER Diagram for the project:



Create project

- Create a working directory, name it as you wish, cd to new directory.

- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv env -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign before your command prompt.
- Install django:

```
pip install django
```

- See installed packages:

```
pip freeze

# you will see:
asgiref==3.3.4
Django==3.2.4
pytz==2021.1
sqlparse==0.4.1

# If you see lots of things here, that means there is a problem with your virtual
env activation.
# Activate scripts again
```

- Create requirements.txt same level with working directory, send your installed packages to this file, requirements file must be up to date:

```
pip freeze > requirements.txt
```

- Create project:

```
django-admin startproject main .
# With . it creates a single project folder.
# Avoiding nested folders
```

- Various files has been created!
- Check your project if it's installed correctly:

```
python3 manage.py runserver  # or,
python manage.py runserver  # or,
py -m manage.py runserver
```

Secure your project

.gitignore

- Add standard .gitignore file to the project root directory.
- Do that before adding your files to staging area, else you will need extra work to unstage files to be able to ignore them.

python-decouple

- To use python decouple in this project, first install it:

```
pip install python-decouple
```

- Update requirements.txt:

```
pip freeze > requirements.txt
```

- For more information look at python-decouple documentation
- Import the config object on settings.py file:

```
from decouple import config
```

- Create .env file on root directory. We will collect our variables in this file.

```
SECRET_KEY=t5o9...
```

- You can use django secret key generator apps
- Retrieve the configuration parameters in settings.py:

```
SECRET_KEY = config('SECRET_KEY')
```

- From now on you can send you project to the github, but double check that you added a .gitignore file which has .env on it.

Publish your project to Github

- The easiest way to publish your local project to github is using the VSCode extention.
    - Open Source Control page and click "Publish to Github".

    - Write a descriptive name and publish.
    - Push your code, set main as upstream (remote) branch name.
    - If you want, go to your Github page and change visibility or your project to "Public".
- Create a remote repo in your Github account first and follow descriptions on the page.

PostgreSQL setup

- To get Python working with Postgres, you will need to install the “psycopg2” module.

```
pip install psycopg2
```

- Update requirements.txt:

```
pip freeze > requirements.txt
```

Django Rest Framework (DRF) setup

- Get the latest installation DRF link
- Install DRF:

```
pip install djangorestframework
```

- Update requirements.txt:

```
pip freeze > requirements.txt
```

- Add 'rest_framework' to your INSTALLED_APPS setting.

```
INSTALLED_APPS = [
    # ...
    # Third party apps:
    'rest_framework',
]
```

Install Swagger

- Explain a sample API reference documentation

- Swagger is an open source project launched by a startup in 2010. The goal is to implement a framework that will allow developers to document and design APIs, while maintaining synchronization with the code.
- Developing an API requires orderly and understandable documentation.
- To document and design APIs with Django rest framework we will use drf-yasg which generate real Swagger/Open-API 2.0 specifications from a Django Rest Framework API. You can find the documentation here.
- Installation:

```
pip install drf-yasg
```

- Update requirements.txt:

```
pip freeze > requirements.txt
```

- Add 'drf_yasg' to your INSTALLED_APPS setting.

```
INSTALLED_APPS = [
   # ...
   'django.contrib.staticfiles',
   # required for serving swagger ui's css/js files

   # Third party apps:
   'drf_yasg',
   # ...
]
```

- Here is the updated urls.py for swagger. In swagger documentation, those patterns are not up-to-date. Modify urls.py:

```
from django.contrib import admin
from django.urls import path

# Three modules for swagger:
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi


schema_view = get_schema_view(
    openapi.Info(
        title="Flight Reservation API",
        default_version="v1",
        description="Flight Reservation API project provides flight and
reservation info",
        terms_of_service="#",
        contact=openapi.Contact(email="rafe@clarusway.com"),  # Change e-mail on
this line!
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=[permissions.AllowAny],
)


urlpatterns = [
    path("admin/", admin.site.urls),

    # Url paths for swagger:
    path("swagger(<format>\.json|\.yaml)",
schema_view.without_ui(cache_timeout=0), name="schema-json"),
    path("swagger/", schema_view.with_ui("swagger", cache_timeout=0),
name="schema-swagger-ui"),
    path("redoc/", schema_view.with_ui("redoc", cache_timeout=0), name="schema-
redoc"),
]
```

- Migrate:

```
python3 manage.py migrate  # or;
python manage.py migrate  # or;
py manage.py migrate
```

- After running the server, go to swagger page and redoc page of your project!

Install Debug Toolbar

- The Django Debug Toolbar is a configurable set of panels that display various debug information about the current request/response and when clicked, display more details about the panel's content.

- See the Django Debug Toolbar documentation page.

- Install the package:

```
pip install django-debug-toolbar
```

- Update requirements.txt:

```
pip freeze > requirements.txt
```

- Add "debug_toolbar" to your INSTALLED_APPS setting:

```
INSTALLED_APPS = [
    # Third party apps:
    # ...
    "debug_toolbar",
    # ...
]
```

- Add django-debug-toolbar's URLs to your project's URLconf:

```
from django.urls import include

urlpatterns = [
    # ...
    path('__debug__/', include('debug_toolbar.urls')),
]
```

- Add the middleware to the top:

```
MIDDLEWARE = [
    "debug_toolbar.middleware.DebugToolbarMiddleware",
    # ...
]
```

- Add configuration of internal IPs to "settings.py":

```
INTERNAL_IPS = [
    "127.0.0.1",
]
```

Seperate Dev and Prod Settings

- When we start to deploy our Django application to the server or develop a Django application with the team, settings will be a serious problem.
- There is no built-in universal way to configure Django settings without hardcoding them. But books, open-source and work projects provide a lot of recommendations and approaches on how to do it best. Let's take a brief look at the most popular ones to examine their weaknesses and strengths.

First Solution: Keeping local settings in "settings_local.py"

- This is the oldest method. I used it when I was configuring a Django project on a production server for the first time. I saw a lot of people use it back in the day, and I still see it now. The basic idea of this method is to extend all environment-specific settings in the settings_local.py file, which is ignored by VCS.

  - Pros: Secrets not in VCS.
  - Cons: settings_local.py is not in VCS, so you can lose some of your Django environment settings. The Django settings file is a Python code, so settings_local.py can have some non-obvious logic. You need to have settings_local.example (in VCS) to share the default configurations for developers.

Second Solution: Separate settings file for each environment

- This is an extension of the previous approach. It allows you to keep all configurations in VCS and to share default settings between developers.
- In this case, you make a settings package with the following file structure:

```
settings/
   ├── __init__.py
   ├── base.py
   ├── ci.py
   ├── local.py
   ├── staging.py
   ├── production.py
   └── qa.py
```

- We prefer the second solution.
- In the "main" directory, create a new folder named "settings".
- Inside "settings" folder, create;
  - `__init__.py` which is the required file to create a python module.
  - `base.py` which will include common settings.
  - `dev.py` which will include developmend specific settings.
  - `prod.py` which will include production specific settings.
- Copy all the staff inside `settings.py` to `base.py`. And delete `settings.py`.
- `base.py` will be:

```
"""
Django settings for main project.
```

```python
Generated by 'django-admin startproject' using Django 4.0.2.
For more information on this file, see
https://docs.djangoproject.com/en/4.0/topics/settings/
For the full list of settings and their values, see
https://docs.djangoproject.com/en/4.0/ref/settings/
"""

from pathlib import Path
from decouple import config

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/4.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = config("SECRET_KEY")


ALLOWED_HOSTS = ['*']


# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # third party
    "rest_framework",
    "drf_yasg",
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

ROOT_URLCONF = "main.urls"

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]

WSGI_APPLICATION = "main.wsgi.application"


# Password validation
# https://docs.djangoproject.com/en/4.0/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME":
"django.contrib.auth.password_validation.UserAttributeSimilarityValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME":
"django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]


# Internationalization
# https://docs.djangoproject.com/en/4.0/topics/i18n/

LANGUAGE_CODE = "en-us"

TIME_ZONE = "UTC"

USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.0/howto/static-files/

STATIC_URL = "static/"

# Default primary key field type
# https://docs.djangoproject.com/en/4.0/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"
```

- `dev.py` will be:

```
from .base import *


THIRD_PARTY_APPS = ["debug_toolbar"]

DEBUG = config("DEBUG")

INSTALLED_APPS += THIRD_PARTY_APPS

THIRD_PARTY_MIDDLEWARE = ["debug_toolbar.middleware.DebugToolbarMiddleware"]

MIDDLEWARE += THIRD_PARTY_MIDDLEWARE

# Database
# https://docs.djangoproject.com/en/4.0/ref/settings/#databases
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}

INTERNAL_IPS = [
    "127.0.0.1",
]
```

- `prod.py` will be:

```
from .base import *

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        "NAME": config("SQL_DATABASE"),
        "USER": config("SQL_USER"),
        "PASSWORD": config("SQL_PASSWORD"),
        "HOST": config("SQL_HOST"),
        "PORT": config("SQL_PORT"),
        "ATOMIC_REQUESTS": True,
    }
}
```

- `__init__.py` will be:

```python
from .base import *


env_name = config("ENV_NAME")

if env_name == "prod":

    from .prod import *

elif env_name == "dev":

    from .dev import *
```

- Modify `.env` file with environment name, postgres and debug variables:

```
ENV_NAME=dev

DEBUG=True

SQL_DATABASE=fradb

SQL_USER=postgres

SQL_PASSWORD=postgres

SQL_HOST=localhost

SQL_PORT=5432
```

- Migrate the latest changes:

```
python3 manage.py migrate  # or;
python manage.py migrate  # or;
py manage.py migrate
```

Logging

Python programmers will often use print() in their code as a quick and convenient debugging tool. Using the logging framework is only a little more effort than that, but it's much more elegant and flexible. As well as being useful for debugging, logging can also provide you with more - and better structured - information about the state and health of your application.

Django uses and extends Python's builtin logging module to perform system logging. This module is discussed in detail in Python's own documentation; this section provides a quick overview.

A Python logging configuration consists of four parts:

- Loggers
- Handlers
- Filters
- Formatters

Python defines the following log levels:

- DEBUG: Low level system information for debugging purposes
- INFO: General system information
- WARNING: Information describing a minor problem that has occurred.
- ERROR: Information describing a major problem that has occurred.
- CRITICAL: Information describing a critical problem that has occurred.

An example logging setting may be like:

```
LOGGING = {
    "version": 1,
    # is set to True then all loggers from the default configuration will be
disabled.
    "disable_existing_loggers": True,
    # Formatters describe the exact format of that text of a log record.
    "formatters": {
        "standard": {
            "format": "[%(levelname)s] %(asctime)s %(name)s: %(message)s"
        },
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d}
{message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    # The handler is the engine that determines what happens to each message in a
logger.
    # It describes a particular logging behavior, such as writing a message to the
screen,
    # to a file, or to a network socket.
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "formatter": "standard",
            "level": "INFO",
            "stream": "ext://sys.stdout",
            },
        'file': {
            'class': 'logging.FileHandler',
            "formatter": "verbose",
            'filename': './debug.log',
            'level': 'INFO',
        },
    },
    # A logger is the entry point into the logging system.
    "loggers": {
        "django": {
            "handlers": ["console", 'file'],
            # log level describes the severity of the messages that the logger
will handle.
            "level": config("DJANGO_LOG_LEVEL", "INFO"),
            'propagate': True,
            # If False, this means that log messages written to django.request
            # will not be handled by the django logger.
        },
    },
}
```

Django Settings: Best practices

- Keep settings in environment variables.
- Write default values for production configuration (excluding secret keys and tokens).
- Don’t hardcode sensitive settings, and don’t put them in VCS.
- Split settings into groups: Django, third-party, project.
- Follow naming conventions for custom (project) settings.

This is the end of initial setup. Send this setup to your Github repo. You can use it in your projects.

☺ **Happy Coding!** ✍

Clarusway