
DJANGO 6
FIRST STEPS

Django 6 with Python 3 + Vue integration, Bootstrap 5

Your roadmap to developing web applications in Python with Django begins here

Andres Cruz Yoris

This version was published: 2026-01-26

Posted about the book!

Please help promote this book.

The suggested post for this book is:

I just bought the book "First steps with Django" from @LibreDesarrollo!

Get your copy at:

https://www.desarrollolibre.net/libros/get-started-with-django

About the Author

This book was prepared by Andres Cruz Yoris, Bachelor of Computer Science, with more than 10 years of experience in the development of web applications in general; I work with PHP, Python and client-side technologies such as HTML, JavaScript, CSS, Vue among others; and server-side like Laravel, Flask, Django, and Codeigniter. I am also a developer in Android Studio, xCode and Flutter for the creation of native applications for Android and IOS.

I put at your disposal part of my learning, reflected in each of the words that make up this book, my fourth book on software development, but the first book on web development with Python; along with a framework that is a delight to program that has been Django.

Copyright

No part of this book may be reproduced or transmitted in any way; that is, electronically or by photocopies without the permission of the author.

Foreword

Django is a framework with many options, it allows us to create applications with the language of the moment Python, a modular programming language, with an impeccable syntax that we can also use in a web environment.

It offers us a modular scheme between projects and applications, an important set of packages to extend the already complete framework and with this, the creation of scalable projects.

It is a framework that, by using Python, brings us its benefits of a clean, clear, precise syntax, a modular and scalable environment through packages.

In this book, we are going to learn how the framework is formed, the basic characteristics that Django allows us as the framework par excellence to create CRUDs processes, which range from the creation of views, templates, connection to models, routing, form validations, creating middleware among others.

In short, you will have an enormous material at your disposal, to make true web applications.

Who is this book for

This book is for anyone who wants to start developing with Django; you may come from another framework or even PHP; in both cases, it is a great idea to learn from one of the most advanced and complete web frameworks you can find.

For those who want to know the framework and who know other web frameworks, but do not have the necessary knowledge to venture into these.

For people who want to improve a skill in web development, who want to grow as a developer and who want to keep climbing their way with other frameworks similar to this one.

For those who want to learn or improve a skill and with this, increase their chances of employment or to carry out a personal project.

(The book is currently in development)

Considerations

Remember that when you see the $ symbol it is to indicate commands in the terminal; You do not have to type this symbol in your terminal, it is a convention that helps you know that you are executing a command.

At the end of each chapter, you have the link to the source code so you can compare it with your code.

The use of bold in paragraphs has two functions:
If we place a letter in bold it is to highlight some code such as the name of variables, tables or the like.
Highlight important parts or ideas.

To display tips we use the following layout:

important tips

For the code snippets:

```
$ python -V
```

When using the *** or //***

It means that we are indicating that in the code there are fragments that we presented earlier.

The official page of the framework has been:

https://www.djangoproject.com/

Which is essential to follow this book and with this learn more aspects of the framework.

Errata and comments

If you have any questions, recommendations or found any errors, you can let them know at the following link:

https://www.desarrollolibre.net/contacto

By my email:

desarrollolibre.net@gmail.com

Or by Discord on the Django channel:

https://discord.gg/sg85Zgwz96

As a recommendation, before reporting a possible problem, check the version to which you refer and add it in your comment; it is found on the second page of the book.

Introduction

This guide is intended to get you started with Django; with this, we are going to establish two things:
It is not a book that aims to know 100% Django, or from zero to expert, since it would be too big an objective for the scope of this guide, if not to know its ecosystem, what it offers us and how it works in based on various examples and/or small applications with limited scope.

It is assumed that the reader has at least basic knowledge of the Python programming language, as well as web technologies such as HTML, CSS, and JavaScript; it is also recommended that you have at least theoretical knowledge about the framework in the use of modules, migrations, MTV; although in the book we make introductions to the entire base structure of the framework.

Map

Chapter 1 - **Required software and installation of tools**: In this chapter we are going to present the software necessary to create projects in Django, Python as a programming language and in which the framework is developed, a code editor such as VSS, until preparing the environment.

Chapter 2 - **Getting started with Django**: In this chapter we are going to create the virtual environment to develop the application, we will create the project in Django, we will learn about its basic structure and the use of applications and projects in Django.

Chapter 3 - **Introduction to Python**: In this chapter we will give an introduction to Python, knowing its basic characteristics and functionalities such as variables, data types, functions, classes, among others.

Chapter 4 - **Create a CRUD application**: In this chapter, we are going to create a simple CRUD type application and learn about all the elements of the framework that this implies such as management of forms, templates, views and models, in addition to the management of CSRF tokens, migrations, databases among others.

Chapter 5 - **Django Admin**: In this chapter, we are going to work with Django Admin to provide an interface for managing site content; that is, the typical CRUDs, we will learn how to customize them at the class level, validations, interpret requests, among others.

Chapter 6 - **Forms**: In this chapter, we are going to introduce the use of forms in Django, definition, validations, saving and structure in general.

Chapter 7 - **Templates: Master Templates, Tags, and Filters**: In this chapter, we are going to present the different functionalities that templates have in Django.

Chapter 8 - **ORM - Working with QuerySets**: In this chapter, we are going to introduce some functions that we can use with QuerySets when building queries.

Chapter 9 - **Create a Rest API with Django Rest Framework**: In this chapter, we are going to build a Rest API with the Django Rest Framework.

Chapter 10 - **Consuming the Rest API with an app in Vue** : In this chapter, we are going to consume the above Rest Api with an app in Vue 3 and NaiveUI; lists, forms will be created and integrated with Tailwind.css

Chapter 11 - **Class Based Views (CBV)**: In this chapter, we are going to learn about class-based views by performing the typical operations that are needed in modern web development.

Chapter 12 - **Manage files**: In this chapter, we are going to learn how to work with files in Django; specifically reading and generating excel and CSV and generating PDFs.

Chapter 13 - **Create an Alpine.js to-do list application and integrate it with Django**: In this chapter, we are going to show the code of an application created with Alpine.js to create a to do list component and we are going to adapt the Alpine.js application presented in the previous chapter, so that it works together with Django.

Chapter 14 - **Django Channels**: In this chapter, we are going to use Django Channels to learn how to create systems with full duplex communication and be able to create our own websockets and consume them, we will see how to create single and multiple channel applications, as well as how to consume a consumer from an app in Vue.

Chapter 15 - **Custom authentication**: In this chapter, we are going to create a personalized authentication module with login, registration, recover passwords, sending emails, uploading the avatar among others.

Chapter 16 - **Introduction to Testing**: In this chapter, we will introduce the use of testing to create automated functions and test different modules of our application.

Chapter 17 - **General Aspects**: In this chapter, we'll learn how to use the signals system to intercept events in Django, such as sending emails, translations, sessions, and more.

Chapter 18 - **Essential Packages**: We'll look at a list and how to use essential packages in Django. These packages are very interesting and can facilitate all kinds of development, from configuring automatic thumbnails for our images, installing a toolbar for debugging, exporting to CSV, customizing Django Admin, generating test data, and more.

Chapter 19 - **Deployment**: We will see how we can deploy the project to the Railway service.

Index

- **Chapter 1: Required software and installation of tools**
  - Required Software
    - Python
    - Visual Studio Code
    - Browser
    - Database: MySQL
  - Questions
- **Chapter 2: Getting started with Django**
  - Packages and best practices when working with Python
    - Virtual environments
    - In summary, what do virtual environments allow us to do?
  - Prepare the development environment
    - Install virtual environment
    - Create virtual space manually
    - Optional VS Code: Creating the Virtual Environment using VS Code
    - Optional VSC: Select Interpreter
  - Django
    - Key Features of Django

        - The "Batteries Included" Concept
        - Support and Non-Disruption
        - Additional Scalability
        - About Django
        - Django's Modular Architecture (MVT)
        - Install Django
        - Check the installed Django version
        - Projects and applications
        - Create a project in Django
        - Project structure
        - Command line
        - Project settings
        - Create an application
        - Register application in the project
        - Extra: Cloning a Django Project with Git and Github and Configuration
    - Questions

- **Chapter 3: Introduction to Python**
    - Interactive console/Python console
    - Python files
    - Variables and data type
    - Mathematical operations
    - String concatenation

- Castings
- Lists
- Conditionals
- Loops
- Functions
  - Parameters
- Dictionaries
- Classes
  - Inheritance
- Modules in Python
  - Create a module
  - Import a module
- Questions

- **Chapter 4: Create a CRUD application**
  - Models
    - Database Connection
    - Create models
  - Optional: Gemini CLI and/or Gemini Agent, Your AI Assistant
    - 1. Gemini Code Assistant (VS Code Extension)
    - 2. Gemini CLI (Command Line Interface Tool)
    - 3. Error Identification and Correction
  - Introduction to migrations

    - Migrations
    - Benefits of Using Migration
    - Create migrations
    - Run migrations
    - Revert migrations
    - Other operations related to migration
    - Other Migration Operations
    - Do Migrations Make Sense?
- Hello World
- Run the project on a server
    - Django Responses
    - The Django Development Server
- Views, Templates, and Routes
    - Views
    - Templates
    - Routes
- Views, templates, routes and forms
    - Pass parameters to template
    - Container folders for templates
    - Forms
    - Getting the Data from the request Object
    - Data Persistence: Saving with the Model

    - ModelForm: Forms for Models
    - Process forms
- Interact with the Database and CRUD Operations
    - Introduction to CRUD and ORM
    - Models
- Shell
    - ???? Django Shell vs. Python Shell
- Create a CRUD process
    - List
    - Edit
    - Delete
- Important Considerations
    - A. The Need for a Valid HTTP Response
    - B. Missing Object (Non-existent ID)
    - C. Paginated Listings (Pagination)
- Redirects
    - 1. The Problem of Returning the Page Directly
    - 2. The Solution: The PRG (Post/Redirect/Get) Pattern
- Pagination
    - 1. Pagination Classes
    - 2. Pagination Logic in the View (index)
    - 4. Template

        - Find or 404: Detail
        - Find or 404: List
    - Faker
        - Why Generate Test Data?
        - Generate test data
        - Script to generate test data
    - Optional: Git and GitHub
        - Why Version Control?
        - Repository Configuration on GitHub
        - Preparing the Project for Git
        - Initial Upload to GitHub
    - Extra: Prototyping Layout with A.I. Assistance
    - Questions

- **Chapter 5: Django Admin**
    - Create superuser
        - Advantages of Superuser Access
    - Django Admin in practice
        - Add our own relations
    - Customize Django Admin
        - Define columns for lists
        - Search fields
        - Navigation by dates

        - Ordering
        - Filtered
        - Editable fields
        - Indicate manageable fields
        - Exclude manageable fields
        - Register classes using a decorator
        - Register the model for types, categories and elements
        - Group management fields
        - Form: Customizing Fields
        - List View: Creating Fields
        - Customizing Actions and Buttons in Forms
        - Inline Forms (Inlines) for Relationships
        - Customizing/Overriding CRUD Management Methods
        - Custom Validations in Models and Forms
        - Automation: Generate data for form fields automatically (slug)
        - Theme and static files CSS, JS...
    - Questions

- **Chapter 6: Forms**
    - Two Types of Forms in Django: ModelForm vs. Form
        - 1. ModelForm (Model-Bound Forms)
        - 2. Form (Standard Forms)
    - Django ModelForm

- Django Forms
- Case study
  - Form
  - View for the form
  - Demo: equivalent of the form with ModelForm
  - Elements CRUD
- Validations
  - Show validation errors
  - Custom validations
- Reference the fields and labels of the form individually
- Form Assets (Media class)
  - Customization and Styles in Django: The Widget Mechanism
  - 2. Customization with Widgets locally
  - Widget attribute
  - 3. Global Customization with Widgets
  - 3. Global Customization with Widgets
  - Critique and Recommendation
  - Widget class: create our own widgets
  - Modify attributes
  - Field types
  - Django Widgets
- Field group

- Questions

- **Chapter 7: Templates: Master Templates, Tags, and Filters**
  - The Role of the Template
  - Representing Dynamic Content
    - How to Use Python Functions
  - Key Elements of the Template Engine
  - Render dynamic content
  - Tags
    - Master Template
    - Logical tags
    - Program our own tag
  - Filters
    - cut
    - date
    - now
    - lower
    - upper
    - add
    - with
    - cycle
    - default
    - length

        - safe
        - join
        - Program our own filters
    - Changing Template Engine - Jinja
    - Management and Loading of Static Files (CSS, JS, Images)
        - 1. The Responsibility of Serving Static Files
        - 2. Configuration and Structure
        - 3. Loading in the Template
    - Questions

- **Chapter 8: ORM - Working with QuerySets**
    - CRUD operations using the shell
        - Create an object
        - Get an object
        - Update an object
        - Retrieving objects
        - Filter objects
        - Exclude results
        - Order
        - Object delete
        - 1. CRUD Operations Summary
        - 2. Read and Filtering Summary (QuerySet)
        - 3. Special Search Operators Summary (Lookups)

        - 4. Ordering Summary (order_by)
    - Logical Operators in QuerySets: AND, OR, and NOT
        - 1. The Logical AND Operator
        - 2. The Q Class for OR and NOT
    - Demonstrating the N + 1 Problem
        - Where is the problem?
        - What happens internally
        - Why the N + 1 problem occurs
        - When it is a problem
        - How to detect the problem
        - Solution: select_related
        - Another solution: making it global from the model
        - Lazy Loading vs Eager Loading
    - Reverse Relations in Django
        - Accessing the default _set
        - Customization with related_name
        - The N+1 Query Problem
        - Optimization with prefetch_related
    - Questions

- **Chapter 9: Create a Rest API with Django Rest Framework**
    - What is Django REST Framework (DRF)?
        - What is a REST API?

    - Server and Client
    - What a REST API Can Do
    - REST and its Rules
    - HTTP and Methods
    - APIs in General
    - Why Django REST Framework?
    - Response Formats: JSON and XML
    - API Consumption
- Create a Rest API in Django
- Install
- Create application for the Rest API
    - Project Structure: API Application Creation
- Django Rest Framework in practice
    - 1 Serialization
    - 2 Viewset
- API Implementation: Serializers, ViewSets, and Routes
    - Serialization
    - ViewSets
    - Routes
    - Testing with the REST API through its interface or automated documentation
    - Enable pagination
    - Customize Serializations

        - Customize Viewsets
        - Hook Methods in Viewsets
    - Authentication
        - The Stateless Principle
        - Global Authentication and Permissions Settings
        - Local Configuration (Per ViewSet)
        - Session or basic authentication
        - Token authentication
    - Configure CORS
        - Why is CORS necessary?
        - Package Installation and Configuration
    - Extra: Automatic Slug Generation
        - Boosting development with AI
    - Creating REST APIs in Django with Django Ninja
        - Main Features of Django Ninja
        - Project Structure and Schemas
        - Authentication with JSON Web Tokens (JWT)
        - Which Package to Choose?
    - Questions

- **Chapter 10: Consuming the Rest API with an app in Vue**
    - Development Tools: Node.js and NPM
        - Why use Node.js instead of a CDN?

    - Installing Node.js
- Organizing a project in Django
- Additional resources to learn Vue
- Creating the Project with Vue NPM and Vite
    - VSC
    - Vue.js Project Structure
    - The src/ folder: The heart of the application
    - Installing Vue Dependencies and node_modules
- Vue Router
    - What is Vue Router for?
    - Installation
    - Configuration
    - Vue Router Configuration
- Install and configure axios
    - 1. How do two independent applications communicate?
    - 2. Why do we choose Axios instead of Fetch?
    - 3. Installation from the Node ecosystem
    - 4. Global Configuration in Vue
- Phase 1: Listings
    - API Consumption with Axios and Vue.js
    - 1. Environment Preparation
    - 2. Component Logic: The <script> block

    - 3. The Life Cycle: The mounted() method
    - 4. Asynchronous Requests with Axios
    - 5. Communication between Script and Template
    - 6. Verifying the Request in the Browser
    - 7. What is JSON?
    - 8. Debugging with Breakpoints
    - 9. The Life Cycle: mounted()
    - 10. Data Rendering: The v-for Directive
    - 11. Reactivity and Development Tools
    - 12 Navigation in SPA Applications: href or RouterLink?
- Comparison: Axios vs. Fetch API
    - 1. Availability and Support
    - 2. Syntax and Verbosity
    - 3. Nested Promises
    - 4. Error Handling with catch
    - Summary
- Naive UI, for graphical interface components
    - Advantages of Naive UI over traditional Bootstrap
    - Exploring the available components
    - Install
    - Set up
    - Configure table in lists

    - Container: n-space
    - Layout
    - Menu
    - Menu: Navigation via Router Link
- Install and configure Tailwind CSS 4
    - 1. Tailwind CSS Installation (Version 4)
    - 2. Configuring the "Driver" in Vite
    - 3. Style Importation
    - 4. Verification: The Style "Reset"
    - Fix small details
    - The @apply directive
    - Element list
    - Grouped Routes
    - Route for the list of elements from the layout
    - List of elements by category
    - Element list by types
    - Element Detail
- Extra: Introduction to overriding the NaiveUI theme
    - Common Styles vs. Specific Styles
    - How to find which properties to change?
    - Final Result
- Step 2: CRUDs and forms

        - Creating Forms with Vue and Naive UI for the categories
        - 1. The Form Component in Naive UI
        - 2. Submission Logic and Error Handling
        - 3. Vue and Naive UI Directives
        - 4. Data Structure
        - 5. The Workflow
        - Creating a category with server-side validations
        - Reactivity and Form Tests in Vue
        - Edit a category
        - Links for CRUD in the categories
    - CRUD for type
    - CRUD for Elements
        - Form Structure and New Fields
        - Data Mapping to SELECTs
    - CKEditor: Editor for Rich Content
        - Why CKEditor?
        - Integrating Vue with CKEditor
        - Customization with the "Online Builder"
        - Enable CSS for CKEditor
    - Navigable menu options
    - Format of the dates in the rest api
    - Redirects on nonexistent routes

- Style and Layout Improvement with Tailwind CSS
  - 1. The process with the Assistant (Gemini)
  - 2. Global Structure and Form Design
  - 3. The Grid System and Responsiveness
- Multiple router-views
- Bonus: Using Aliases (@) in Imports
- Questions

- **Chapter 11: Class Based Views (CBV)**
  - 1. What exactly are CBVs?
  - 2. Advantages: Less Code, Better Readability
  - 3. CBV vs. FBV: When to use each?
  - 4. Types of Class-Based Views
  - Create a project and application
  - Create model, migration and form
  - Class-Based Views
    - Form View Class: FormView
    - Overview Class: View
    - Routes
    - Template
    - Test the app
    - TemplateView
  - CRUD class-based views

        - UpdateView: Update
        - CreateView: Create
        - DeleteView: Delete
        - DetailView: Detail
        - IndexView: List
    - Class-based views for authentication
        - Create an app
        - LoginView: View for the login
        - LogoutView: View for logout
        - PasswordChangeView: View to change the password
        - PasswordChangeRest: View to change password reset
    - Questions

- **Chapter 12: Manage files**
    - Create a project and application
    - CSV handling
        - CSV files
        - Before starting
        - File management
        - Read a file
        - Read a file in dictionary format
        - Write a file
        - Write a file in dictionary format

        - Reserved word with
    - Excel handling (XLSX)
        - We create the application
        - Write an excel
        - Read an excel
    - Generate PDF
        - We create the application
        - Generate a PDF from an HTML page
        - Generate a PDF from a Canvas
        - Download any file
    - Questions

- **Chapter 13: Create an Alpine.js to-do list application and integrate it with Django**
    - Why Alpine.js?
        - Basic Concepts and Directives
    - To Do List Application with Alpine.js
        - Packages Used:
        - Code Structure
    - Adapting the To Do List app to Django
        - Create virtual space, project and initial application
        - Migrate to do list app to Django
        - Create model and migration for the To Do
        - Create a Rest API for the To Do

        - Install axios
        - Consuming the To Do list from the Rest API
        - Create to do from Django
        - Edit a to do
        - Delete a to do
        - Mark completed a to do
        - Reordering based on IDs
        - Show to do ordered
        - Delete all to do
        - Protect with authentication required and make adaptations
    - Questions

- **Chapter 14: Django Channels**
    - Django Channels and WebSockets
        - The Problem with Traditional "Real-Time"
        - The Solution: Django Channels and WebSockets
        - WSGI and ASGI Servers
        - Synchronous Applications using WSGI (Web Server Gateway Interface)
        - What is Full Duplex and why do we need it?
        - Asynchronous Applications using ASGI (Asynchronous Server Gateway Interface)
        - WSGI: Classic/Synchronous Connection
        - ASGI: Asynchronous/Synchronous and Full-Duplex Connection
        - Summary

- Set up project in Django with Django Channels
  - Basic installations and configurations
  - Configure ASGI server and Django Channels
  - Basics for our app: Rooms and messages/alerts
- Websockets, Consumers, Asynchronous and Client (View and Template) - (Single channels)
  - What are Consumers in Django Channels?
  - Key Concepts: WebSocket, Consumer, and Asynchrony
  - Client: Consume Websocket
  - Download and Installation of Static Files, Bootstrap 5
- Client: Send and receive messages
  - Flow and Validation: Why Does the Message Bounce?
- Communication through channel layers (Multiple channels)
  - Why not use traditional HTTP?
  - The Concept of Layers: Channel Layers
  - Redis
  - Key concepts
- Implementing Multiple Channels in Django: Changes to the chat app
  - The bridge between Synchronous and Asynchronous Code
  - Client changes
  - Real-Time Demonstration
  - Conclusion: Single Channel vs. Multiple Channels
  - Convert consumer to asynchronous code

- DRF, Django Channels and Vue: Alert Messages Application
  - Django
  - Vue
- Using multiple Consumers
  - The Dual Authentication problem
  - Solution: Grouping Middlewares
  - How it works
- Extra, AI Changes: Auto-scroll, Reloading the list (Ref), and CSS Effect
  - 1. Auto-scroll to the bottom of messages
  - 2. Component Communication (Child to Parent)
  - 3. References and Methods (refs)
  - 4. Animations and Transitions
  - AI and Git Workflow
- Next step, learn FastAPI (and other changes made)
  - Frontend Changes (Vue.js)
  - Conclusion
- Application Redesign with AI: Google Antigravity IDE
  - Tools and Preparation
  - Applying the Redesign
  - Analysis of Results
- Questions

- **Chapter 15: Custom authentication**

- Create project and application
- Expose routes for user authentication
- Building the Login Form
- Install Bootstrap 5 and Font Awesome
- Master template
- Specifying HTML attributes with django-widget-tweaks
- Authentication templates
- Layout of the remaining authentication templates
    - Priority: Django Admin vs. Custom Templates
- Register a user
    - Custom Registration Form to register your email
- Send emails
    - Recovery Flow Steps
- User profile
    - Upload avatar
    - Other fields
    - User profile
- MEDIA_URL and MEDIA_URL: URL Configuration in Development
    - 1. document_root
    - 2. settings.MEDIA_URL
    - The "Virtual Folder": MEDIA_URL
- Questions

- **Chapter 16: Introduction to Testing**
  - Why test?
  - Why Learn Testing in the Age of AI?
  - What are tests for?
    - What should we try?
    - Limitations of Testing
    - Understanding the first Tests
  - Test Django applications
    - The Client object: Your internal browser
    - The Limitations of the 200 Status Code
    - TestCase classes
    - Tools for testing
  - Test context and test database.
    - The setUp method: Context Initialization
    - Test Database
    - Test Database Configuration
    - Atomicity and Test Independence
    - Conclusion
  - Comments Crud
    - Setting Up the Test Environment
    - Index: Pagination
    - Post
    - Update

    - Delete
    - Conclusion of the first CRUD
    - Other assertion methods
    - Modular Testing: Beyond the MTV Pattern
    - Forms
- Modularization of Tests: Improve file and folder structures
    - ??Donde ubicar las pruebas de la API?
- Api
    - Why use the "API" versions?
    - Paginated comments
    - Detail
    - Create
    - Update
    - Delete
    - Serialization
- Elements API
- Generating tests with AI - Gemini Agent
    - Defining the Prompt for Gemini
    - Generating tests for views and forms
    - Final Recommendations
    - Pagination
    - Create

        - All the tests
    - Authentication
        - Creating Users with the Django ORM
        - Authentication in Tests: Sessions and Tokens
    - Factory
    - Conclusions on AI and Unit Testing
    - Questions

- **Chapter 17: General Aspects**
    - Sending Emails with HTML Content
        - 1. Basic test with HTML tags
        - The send_mail method
        - Using Templates for the Email
        - Integration with Signals
        - Final Integration in the View
    - Cryptography in Django
        - Difference between Signing and Encryption
        - The importance of the SECRET_KEY
        - How to sign a String with Signer
        - Recovering the original value (unsign)
        - Variations: The use of "Salt"
        - Protection against alterations
        - Use Cases: When NOT to use Signatures?

- Conclusions
- Cryptography and Encoding: Use Cases in Django
  - 1. Reversible Encoding: Base64 and Encryption (To hide text)
  - 2. Hashing with Python and Django (Irreversible)
  - 3. Hashing with Django (Irreversible)
  - Hash Verification and Advanced Security
  - Summary of Concepts
- Session
  - How to Save, Read, and Delete Data in the Session
  - Limitations: What can we store?
- Using Flash Messages in Django
  - When to use them?
  - 1. Configuration and Message Types
  - 2. How to send a "Flash Message"
  - 3. How to display it in the Template
  - Sending multiple messages
  - 4. Difference with request.session
- Setting up MySQL in Django
  - Connectors for each database engine
  - Driver Selection: mysqlclient vs. PyMySQL
  - mysqlclient, the official driver
  - Testing the UNOFFICIAL version of the MySQL connector

    - Configuring the settings.py file
- Log to file
    - Basic Implementation
- What is Throttling and why use it in Django Rest Framework?
    - Global Configuration and Throttling Types
    - Defining Limits (Rates)
    - Global vs. Local Throttling
    - Local Implementation (Recommended)
    - Testing and Error Messages
    - Advanced Considerations
- Modern Programming: Using AI with Google Antigravity IDE
    - The Project: Blog Module with Django
    - The Planning Tool
    - Implementation and Results
    - Conclusions
- Introduction to Celery: Background Processing in Python: FastAPI, Django, and Flask
    - How does it work?
    - Key Concepts: Broker and Backend
    - Installation and Basic Commands
    - Practical Implementation (FastAPI Example)
    - Practical Implementation (Flask Example)
    - Practical Implementation (Django Example)

        - Status Verification
        - Conclusion
    - Questions

- **Chapter 18: Essential Packages**
    - Django Admin Customization: Aesthetics and Modernization
        - Third-Party Packages for the Admin
        - Choosing the Ideal Package: Analysis and Comparison
        - Legacy: Installation and Configuration Step by Step
        - Final Result
        - Django Unfold Installation and Configuration
        - Result and Conclusion
    - Introduction to Easy Thumbnails
        - Installation and Configuration
        - Dimension and Quality Configuration
        - Usage in Templates
        - Implement
        - Migration and Storage Management
    - Introduction to Django Debug Toolbar
        - Project Configuration
        - Usage and Query Analysis
        - Panel Customization
    - APIs with Django Rest Framework

- Django CORS Headers
  - Usage and Application
- Django Channels and Real-Time Communication
- Implementation of Consumers and WebSockets
  - Interconnectivity with other Technologies
  - Conclusion
- Form Customization with Django Crispy Forms
  - Implementation in the Project
- Django Import Export
  - Installation and Configuration
  - Creating Resources
  - Integration with Django Admin
  - Export/Import from any view
  - Import and Export Processes
  - Data Import
  - Custom Export
- Generating Test Data in Django
  - Installation and Configuration
  - Using the seed Command
  - Possible Compatibility Issues
  - Create Script
- Django Filters

        - Installation and Configuration
        - Creating the FilterSet
        - Implementation in Django REST Framework (DRF)
        - Implementation in Traditional Views (Templates)
    - Django Environment Variable Configuration
        - Use Case: Cross-System Compatibility
        - Installation and Configuration
        - Creating the .env file
        - Configuration in settings.py
        - Security and Git
        - Tip: What to do if it doesn't work?
    - Questions

- **Chapter 19: Deployment**
    - Railway
        - Usage
        - Viewing the repository from Github
        - Project Requirements
        - SQLite Database
        - Publish project from Github
        - Create domain
        - Project Configurations
        - Static files

    - Optional: Database Configuration
    - Web server
- Pythonanywhere
    - Copy the project
    - Before uploading/cloning
    - Steps using git
    - Create a Virtualenv
    - Install project dependencies
    - Configure the app on PythonAnywhere
    - Next steps
- Docker: Django App
    - Connecting Docker with the Local Machine: Practical Explanation: host.docker.internal
- Questions

Image Index

1. Figure 1-1: Python Web
2. Figure 1-2: Add Python to the Path
3. Figure 2-1: Virtual environments in your operating system
4. Figure 2-2: Virtual environment for projects in Django
5. Figure 2-3: Select interpreter
6. Figure 2-4: Window to select interpreter with Command/Control + Shift + P
7. Figure 2-5: Select Python for the interpreter
8. Figure 2-6: Create virtual environment
9. Figure 2-7: Select venv
10. Figure 2-8: Select work area
11. Figure 2-9: Select Python interpreter
12. Figure 2-10: Project and application
13. Figure 2-11: Application files and folders
14. Figure 3-1: Initial migration for element application
15. Figure 3-2: Initial migration for comment application
16. Figure 3-3: Database File in SQLite
17. Figure 3-4: Created tables
18. Figure 3-5: Hello World
19. Figure 3-6: Rendered Template
20. Figure 3-8: Template not found

21. Figure 3-9: Subfolders in the templates
22. Figure 3-9: Form to submit
23. Figure 3-10: 403, CSRF token does not exist
24. Figure 3-11: Record created
25. Figure 3-12: ModelForm and template
26. Figure 3-13: ModelForm in the template with content
27. Figure 3-14: List of records
28. Figure 3-15: ModelForm in the template with content
29. Figure 3-16: List of comments
30. Figure 3-17: 'page' Parameter in the URL
31. Figure 4-1: Login page
32. Figure 4-2: CRUD of users and groups
33. Figure 4-3: Add a group
34. Figure 4-4: Register a custom model
35. Figure 4-5: Add a comment
36. Figure 4-6: List of comments
37. Image
38. Figure 4-8: Set custom columns in the listing
39. Figure 4-9: Filter by dates
40. Figure 4-10: Filter by ID and date
41. Figure 4-11: Modify comments from the list
42. Figure 4-12: Add a comment

43. Figure 4-13: Exclude option
44. Figure 4-14: Group fields on a single line
45. Figure 4-15: Group fields on a single line
46. Figure 5-1: Form to create elements
47. Figure 5-2: List of tasks
48. Figure 5-3: Validation errors on the server
49. Figure 5-4: Validation errors on the client
50. Figure 5-5: Text field
51. Figure 5-6: Label
52. Figure 5-7: Django Admin Calendar Widget
53. Figure 5-8: Widgets available in Django Admin
54. Figure 8-1: List of comments
55. Figure 8-2: Create a comment
56. Figure 8-3: Update a comment
57. Figure 8-4: Create a comment with Fks
58. Figure 8-4: Basic authentication
59. Figure 8-5: Login using postman
60. Figure 8-6: Consume a resource using Postman
61. Figure 9-1: Initial listing
62. Figure 9-2: Naive UI Table Component
63. Figure 9-3: Naive UI Table Component in Types
64. Figure 9-4: Basic layout

65. Figure 9-5: Basic layout for the application
66. Figure 9-6: Menus in the application layout
67. Figure 9-7: Header with menu
68. Figure 9-8: Padding added with Tailwind.css
69. Figure 9-9: List of elements by type
70. Figure 9-10: Element detail
71. Figure 9-11: Theme editor
72. Figure 9-12: Form for creating a category
73. Figure 9-13: Form for creating an element
74. Figure 9-14: CKEDITOR Editor
75. Figure 10-1: Form to create a book
76. Figure 10-2: Login form
77. Figure 10-3: Password change form
78. Figure 10-4: Password change form, validation errors
79. Figure 10-5: Recover Password
80. Figure 11-1: CSV Data in Excel
81. Figure 11-2: Data in Excel
82. Figure 11-3: Generate PDF
83. Figure 11-4: Generate PDF, Hello World
84. Figure 11-5: Geometric figure
85. Figure 11-6: Change colors line
86. Figure 12-1: To Do App in Alpine

87. Figure 17-1: WSGI Server Process
88. Figure 17-2: ASGI Server Process
89. Figure 17-3: Test Users
90. Figure 17-4: Test rooms
91. Figure 17-5: Bootstrap CDN
92. Figure 17-6: Redis in DBngin
93. Figure 17-5: Laragon
94. Figure 17-6: Enable Redis in Laragon
95. Figure 17-7: Successful login
96. Figure 17-8: Get alerts with the token
97. Figure 17-1: Login Form
98. Figure 17-1: Bootstrap CDN
99. Figure 17-2: Bootstrap and Fontawesome installed in the project
100. Roadmap generated by Google Antigravity
101. Obtained design
102. Image
103. Image
104. Image
105. Image
106. Image
107. Image
108. Image

109. Image
110. Image
111. Image

Questions

- **Required software and installation of tools**
    - According to the text, what is the only program you absolutely need to install on your computer to start programming with Django?
        - **Python**
        - Visual Studio Code
        - MySQL
        - Django
    - What is the recommended browser for development, as mentioned in the text?
        - Firefox
        - Safari
        - **Google Chrome**
        - Microsoft Edge
    - What is the purpose of 'pip' in Python development?
        - To install a code editor
        - **To manage Python dependencies and packages**
        - To create Django projects
        - To connect to a MySQL database
- **Getting started with Django**

- What is the primary benefit of using a requirements.txt file when cloning a Django project?
    - It automatically generates the virtual environment.
    - **It allows for quick and easy installation of all project dependencies.**
    - It provides a backup of the project's source code.
    - It simplifies the deployment process to a production server.

- What is the main advantage of using venv over other virtual environment tools like conda?
    - **Venv is recommended by the official Python documentation and is widely used.**
    - Conda is faster for creating virtual environments.
    - Venv is compatible with all operating systems.
    - Conda provides more advanced package management features.

- What command is used to create a new Django application within a project?
    - python manage.py startproject
    - django-admin startapp
    - **python manage.py startapp**
    - python manage.py createapp

- What is the purpose of the INSTALLED_APPS variable in Django's settings.py?
    - **To list all the installed applications within a Django project.**

    - To define the project's database connection settings.
    - To specify the virtual environment used by the project.
    - To configure the Django admin interface.

- What command is used to activate a virtual environment on Windows using CMD?
    - source bin/activate
    - source Scripts/activate
    - **\Scripts\activate**
    - \Scripts\deactivate

- According to the text, what key combination should be pressed to search for "Select interpreter"?
    - **Command/Control + Shift + P**
    - Command + Shift + P
    - Control + P
    - Command + Control + P

- Why is using a virtual environment recommended when working with Python packages?
    - **It allows you to install packages in isolation from the operating system, avoiding conflicts between different project dependencies.**
    - It automatically installs all necessary dependencies for any Python project.

- It simplifies the process of uninstalling packages from the operating system.
- It eliminates the need to install packages at all.

- **Introduction to Python**
  - What does the 'self' keyword refer to within a class method?
    - **The class itself.**
    - A global variable.
    - The instance of the class.
    - A parameter passed to the method.
  - According to the text, what indicates that the console is waiting for input?
    - '$ python'
    - **'>>>'**
    - 'code'
    - 'print()'
  - What is the purpose of the 'type()' function in Python?
    - To calculate the length of a string
    - To perform mathematical operations
    - **To determine the data type of a value**
    - To convert a value from one type to another
  - What is the purpose of the 'def' keyword in Python?

    - To define a conditional statement.
    - **To define a function.**
    - To create a loop.
    - To create a class.

- What are frameworks primarily used for, as indicated in the text?
    - **Developing standalone applications.**
    - Importing characteristics and packages.
    - Creating operating systems.
    - Designing hardware components.

- According to the provided text, what is one use of importing characteristics and packages within frameworks?
    - **To use the framework's characteristics and associated packages.**
    - To create new frameworks from scratch.
    - To remove unnecessary code from the framework.
    - To simplify the framework's structure.

- What is the first tool recommended for development in Python, according to the text?
    - **The interactive console**
    - Python files
    - A text editor
    - A compiler

- **Create a CRUD application**
    - Why is using `get_list_or_404()` less important than `get_object_or_404()`?
        - **Because an empty list is a desired behavior, while a missing object indicates an error.**
        - Because it's more complex to implement.
        - Because it always returns a 404 error.
        - Because it requires more memory.
    - According to the text, what Django command is used to launch the Django shell?
        - **python manage.py shell**
        - python django shell
        - django shell
        - python manage.py runserver
    - What command is used to create migration files based on Django models?
        - python manage.py runmigrations
        - **python manage.py makemigrations**
        - python manage.py applymigrations
        - python manage.py createschema
    - According to the text, what is the primary purpose of using subfolders within the 'templates' directory?
        - **To avoid naming conflicts and maintain clarity in large applications.**

    - To speed up the rendering process.
    - To simplify the view functions in 'views.py'.
    - To reduce the number of files in the project.

- What is the recommended HTTP method for creating resources, as stated in the text?
    - GET
    - **POST**
    - PUT
    - DELETE

- According to the text, what is the main security problem introduced when saving user-entered data without proper sanitization?
    - **Cross-Site Scripting (XSS) injection**
    - SQL injection
    - Denial of Service (DoS)
    - Brute-force attack

- What is the purpose of the `-m` parameter in the `git commit` command?
    - **It specifies the message related to the commit.**
    - It indicates the branch to commit to.
    - It determines the author of the commit.
    - It defines the commit's parent.

- **Django Admin**

- What does the `save_model` implementation in `app_name/forms.py` ensure?
    - **That the slug will never be saved empty in the database.**
    - That the title is always generated from the slug.
    - That the form always validates successfully.
    - That the slug is always unique.

- What is the purpose of the 'fieldsets' attribute in Django Admin?
    - To group form fields on a single line
    - **To create collapsible blocks of form fields**
    - To define custom widgets
    - To automatically map fields to HTML elements

- What happens when `save_as_continue = False`?
    - The edit form remains open.
    - **The user is redirected to the record list page.**
    - The record is immediately deleted.
    - The form is saved and a new record is created.

- What is the purpose of the 'createsuperuser' command?
    - To start the Django development server.
    - **To create a superuser account for accessing the admin panel.**
    - To automatically generate models for your application.
    - To deploy your Django application to a production server.

- What is the primary purpose of decorators in Django?
    - **To extend or modify behavior without changing source code**
    - To create separate lines of code
    - To directly alter class definitions
    - To simplify model creation

- What is the purpose of the `save_on_top` option?
    - It hides the save buttons.
    - **It duplicates the action button bar in the form header.**
    - It automatically saves the form every few seconds.
    - It disables the save buttons until all fields are filled.

- According to the text, what is one advantage of using decorators over a separate line for model customization?
    - It's less expressive
    - **It's more concise**
    - It requires more complex code
    - It's less readable

- **Forms**
    - According to the text, what is the main advantage of using ModelForms over standard Django forms?
        - **ModelForms automate 90% of the work.**

    - ModelForms provide more control over form fields.
    - ModelForms are easier to implement.
    - ModelForms are required for all Django projects.

- What is the purpose of the `validators` argument when defining form fields?
    - **To apply custom validations to the field's input.**
    - To specify the order of fields in the form.
    - To define the initial value of the field.
    - To connect the field to a specific model instance.

- What is the primary distinction between ModelForm and Form in Django?
    - **ModelForm is tied to a database model, while Form is not.**
    - Form is automatically extracted from the model, while ModelForm requires manual definition.
    - ModelForm offers more flexibility than Form.
    - Form is primarily used for simple CRUD operations.

- According to the text, what is a key advantage of using ModelForm?
    - Greater flexibility in field customization.
    - **Reduced code due to automatic field extraction from the model.**
    - Ideal for complex scenarios involving multiple entities.
    - Faster development speed even for simple use cases.

- According to the text, what benefit is mentioned regarding the newer Django versions (compared to older versions) when rendering forms?

        - More lines of code are required.
        - The process requires manual creation of label tags.
        - **Fewer lines of code are needed.**
        - The email and password fields are automatically hidden.

    - What is the default template used by the `as_field_group()` method?
        - **django/forms/field.html**
        - field_snippet.html
        - my_custom_template.html
        - None of the above

    - What is the purpose of the `Media` class within a widget?
        - To define the widget's attributes.
        - **To specify static resources like CSS and JavaScript.**
        - To render the form field.
        - To handle form validation.

- **Templates: Master Templates, Tags, and Filters**
    - What is the purpose of the 'STATIC_URL' variable in Django's settings.py?
        - **It defines the URL prefix used to reference static files in templates.**
        - It specifies the location of the static files folder.
        - It determines the caching behavior of static files.
        - It sets the permissions for accessing static files.

- What is the purpose of the `{% load custom_tags %}` statement in Django templates?
    - To define a new tag.
    - **To load custom tags from a specific file.**
    - To declare a variable.
    - To import a library.

- What is the primary purpose of the 'with' parameter within the {% include %} tag?
    - **To explicitly pass context variables and name them generically for reusability.**
    - To define custom tags for the template.
    - To create and manage database connections.
    - To define and use blocks in the master template.

- What is the purpose of the 'collectstatic' command in Django?
    - **To gather all static files from different application folders and consolidate them into a single directory.**
    - To automatically generate static files based on templates.
    - To minify static files for faster loading.
    - To deploy static files to a remote server.

- Where should you create a folder called 'templatetags' in a Django project to define custom template tags?
    - **At the root of the Django project.**

    - Within each application directory, at the same level as models.py and views.py.
    - Inside the 'templates' directory.
    - Within the 'settings.py' file.

- Which of the following best describes the purpose of the 'forloop' variable within a Django template's for tag?
    - **To define custom tags for the template.**
    - To provide information about the current iteration of the loop.
    - To establish database connections.
    - To handle logical conditional statements.

- Which syntax is used to print variables in a Django template?
    - {% variable %}
    - **{{ variable }}**
    - {# variable #}
    - <% variable %>

- **ORM - Working with QuerySets**
    - What is the `N+1` problem and how can it be solved using the text provided?
        - **A performance issue caused by excessive queries, solved with select_related.**
        - A database error caused by incorrect filtering, solved with Q objects.

    - A template rendering issue, solved with caching.
    - A model design flaw, solved by restructuring the models.

- According to the text, what class is required to implement OR and NOT operators in Django?
    - **Q class**
    - Filter class
    - Model class
    - QuerySet class

- What is the benefit of using `prefetch_related` as described in the text?
    - **It drastically reduces the number of database queries.**
    - It increases the number of database queries for better performance.
    - It only works with JOINs.
    - It always results in a single query.

- What does Lazy Loading mean in the context of Django ORM?
    - **Relationships are not loaded until they are needed.**
    - Relationships are loaded immediately when accessed.
    - Relationships are always eager loaded.
    - Relationships are loaded only in detail views.

- In the provided example, how many queries are performed when using `prefetch_related('element_set')`?

    - **One query to fetch all categories and one to fetch all related elements.**
    - Two queries: one for categories and one for elements.
    - Three queries: one for categories and two for elements.
    - A variable number of queries, depending on the number of categories.

- According to the text, what is the primary difference between `select_related` and `prefetch_related`?
    - **select_related uses a second separate query, while prefetch_related uses a JOIN.**
    - select_related performs a second separate query, while prefetch_related uses a JOIN.
    - prefetch_related uses a JOIN, while select_related performs a second separate query.
    - Both use JOINs and perform separate queries.

- What does the `__icontains` lookup operator do in Django's ORM?
    - Searches for a substring in a case-sensitive manner.
    - **Searches for a substring in a case-insensitive manner.**
    - Filters based on a greater than condition.
    - Filters based on a less than condition.

- **Create a Rest API with Django Rest Framework**

- What is the primary benefit of using hook methods (perform_create, perform_update, perform_destroy) in ViewSets?
    - **To inject custom business logic before or after serializer saving.**
    - To automatically generate URLs for API endpoints.
    - To handle user authentication and permissions.
    - To optimize database query performance.

- Why is it necessary to create separate serializer classes for elements, one read-only and one write-only?
    - To improve performance of read operations.
    - **To control which fields are serialized during creation/update.**
    - To simplify the code.
    - To provide a different API endpoint.

- What is the primary advantage of token-based authentication mentioned in the text?
    - **It's easier to implement in mobile and web clients.**
    - It provides stronger security than session authentication.
    - It's less resource-intensive than basic authentication.
    - It's automatically integrated with Django Admin.

- What is the primary function of a ViewSet in Django REST Framework?
    - **To define data models.**
    - To handle serialization and deserialization.

        - To group CRUD operations into a single class.
        - To manage user authentication.

    - What is Django REST Framework (DRF) primarily used for?
        - Creating graphical user interfaces.
        - **Building and managing REST APIs in Django.**
        - Database optimization and performance.
        - Testing Django applications.

    - What is the purpose of the 'detail' parameter in the DRF `@action` decorator?
        - **To specify whether the action operates on an individual item or the entire collection.**
        - To define the HTTP methods allowed for the action.
        - To indicate the serializer class to use for the action.
        - To specify the URL name for the action.

    - Which HTTP method is typically used to update an existing resource partially?
        - GET
        - POST
        - PUT
        - **PATCH**

- **Consuming the Rest API with an app in Vue**
    - What is the main advantage of using Tailwind CSS?

    - It provides pre-built components like buttons and tables.
    - **It's a utility-first CSS framework that injects only necessary CSS.**
    - It requires configuring several JavaScript files.
    - It's a framework that relies on pre-built components like Naive UI.

- According to the text, what does the `validation-status` prop in `n-form-item` indicate?
    - The error message to display
    - **The state of the field (warning, success, or error)**
    - The label of the form item
    - The feedback provided by the user

- What problem does using aliases (e.g., @) in imports solve?
    - It makes the code harder to maintain.
    - **It avoids excessively long and repetitive file paths.**
    - It increases the complexity of the project.
    - It slows down the application.

- What is a common need addressed by WYSIWYG plugins in web development?
    - **Incorporating rich content like bold text and lists**
    - Creating server-side rendering
    - Managing database connections
    - Implementing user authentication

- According to the text, why was Vue chosen for this project over other frameworks?
    - **Its short learning curve and flexibility, allowing integration into existing projects.**
    - Its complex syntax and steep learning curve.
    - Its requirement for the entire web page to be governed by it.
    - Its exclusive use for large, complex applications.

- What does the question mark (?) in the route path `/category/:id?` signify?
    - That the category ID is required
    - **That the category ID is optional**
    - That the category ID is a string
    - That the category ID is an integer

- What structure does Naive UI expect for the data in a SELECT element?
    - **{ label: 'Name', value: 'ID' }**
    - { name: 'Name', id: 'ID' }
    - { title: 'Name', value: 'ID' }
    - { value: 'Name', label: 'ID' }

- **Class Based Views (CBV)**
    - What is the primary purpose of using Django's built-in views for authentication?
        - **To simplify the creation of personalized login, logout, recover, and change password functionalities.**

    - To exclusively authenticate superusers.
    - To bypass the need for any authentication system.
    - To integrate with external authentication providers only.

- According to the text, when is it most appropriate to use Function-Based Views (FBVs)?
    - When implementing complex CRUD operations.
    - **When needing a very specific, manual, or complex task that doesn't easily fit a class structure.**
    - When aiming for maximum code cleanliness and readability.
    - When working with Django REST Framework.

- According to the text, what is the purpose of the `redirect_authenticated_user` attribute in the `UserLoginView`?
    - To redirect the user to a different page after logging out.
    - **To automatically redirect an authenticated user after a successful login.**
    - To display an error message if the user is already authenticated.
    - To disable authentication for authenticated users.

- According to the text, what is the purpose of TemplateView?
    - **To return an HTML file (template).**
    - To handle form submissions.
    - To manage database records.

    - To authenticate users.

- What is a key benefit of using Class-Based Views (CBVs) over Function-Based Views (FBVs) in Django?
    - **Increased code reusability and a more structured approach.**
    - They are inherently faster than FBVs.
    - They automatically handle all HTTP request methods.
    - They eliminate the need for templates.

- What is the purpose of the 'as_view()' function when using class-based views?
    - To create a path() function.
    - To create an instance of the class.
    - **To convert the class into a URL pattern.**
    - To validate the form data.

- What is the primary difference between UpdateView and FormView?
    - **FormView is specifically designed to update existing data.**
    - UpdateView requires a custom form.
    - FormView is simpler to use.
    - There is no difference; they are interchangeable.

- **Manage files**
    - What is the purpose of the 'with' statement when working with files in Python?
        - To define an alias for the file operation

        - **To mark a scope where the file object exists, ensuring resources are released automatically**
        - To specify the file encoding
        - To open the file in append mode

    - What package is used to write Excel files in Python, as mentioned in the text?
        - **XlsxWriter**
        - openpyxl
        - csv
        - DictWriter

    - Which package is used to read Excel files, according to the text?
        - XlsxWriter
        - **openpyxl**
        - csv
        - DictWriter

- **Create an Alpine.js to-do list application and integrate it with Django**
    - What command is used to create a new virtual environment?
        - python manage.py venv todo-venv
        - venv create todo-venv
        - **python -m venv todo-venv**
        - pip install venv todo-venv

- What is the significance of using `FormData` instead of a JSON object when sending data to the Django REST API in this example?
    - **FormData is more efficient for sending large amounts of data.**
    - FormData allows Django's `request.POST.get()` method to correctly retrieve the parameters.
    - FormData is required for authentication.
    - FormData is only used for DELETE requests.

- What is the purpose of registering the 'app' in the settings.py file?
    - To install Django.
    - To activate the virtual environment.
    - **To make the application available to Django.**
    - To create a superuser.

- What is the main security flaw described in the text regarding the DELETE method?
    - Lack of CSRF protection
    - **Users can delete any to-do regardless of ownership**
    - Incorrect serializer implementation
    - Insufficient user authentication

- What error would be encountered if the REST API is used from the client (Alpine) when using POST or DELETE requests before the CSRF token is implemented?
    - **CSRF Failed: CSRF token missing.**
    - Authentication Failed.

        - Permission Denied.
        - Invalid Request.

    - What is the purpose of the 'perform_create()' function in the Django REST API?
        - To handle user authentication
        - **To automatically calculate the order of the to-do**
        - To validate the to-do data
        - To display the to-do list

    - How should the header be configured in axios requests using POST, PUT, or DELETE to include the CSRF token?
        - **"X-AuthToken": "{{ csrf_token }}"**
        - "CSRFToken": "{{ csrf_token }}"
        - "X-CSRFToken": "{{ csrf_token }}"
        - "Token": "{{ csrf_token }}"

- **Django Channels**
    - What is the purpose of the `connect()` method within a Django Channels Consumer?
        - It's executed when a message is received from the client.
        - **It's executed when a connection is established with the client.**
        - It's executed when the client disconnects.
        - It's used to send messages back to the client.

- What is the advantage of using a 'ref' instead of changing the key of a Vue component?
    - It causes an annoying visual effect.
    - **It allows direct access to internal methods of the child component from the parent.**
    - It forces Vue to redraw the entire component.
    - It eliminates the need for custom events.

- What is the significance of the 'asgi' application in the context of Django Channels?
    - It's a protocol similar to HTTP, used for full-duplex communication.
    - **It's a Django setting that specifies the ASGI application module.**
    - It's a function that creates a superuser.
    - It's a command for applying database migrations.

- What is the main difference between WSGI and ASGI servers in the context of Django?
    - WSGI servers are asynchronous, while ASGI servers are synchronous.
    - **WSGI servers handle only HTTP requests, while ASGI servers handle HTTP and WebSockets.**
    - ASGI servers are the default in Django, while WSGI servers require additional configuration.
    - WSGI servers are used for real-time applications, while ASGI servers are for standard web requests.

- How does the TokenAuthMiddleware handle authentication when both Django's AuthMiddlewareStack and Vue's token authentication are used?
    - **It prioritizes Vue's token authentication, ignoring Django's session.**
    - It prioritizes Django's session authentication, ignoring the token.
    - It combines both, using Django's session if a token is not found in the URL, and Vue's token if a token is present.
    - It requires a separate authentication mechanism for each application.

- What is the primary benefit of converting the consumer to asynchronous code?
    - **It improves performance and readability of the code.**
    - It requires using async_to_sync more frequently.
    - It simplifies the WebSocket connection process.
    - It eliminates the need for Django Channels.

- What is the purpose of the `connecting` variable in the MessageComponent.vue?
    - **To display a loading message while the websocket is connecting.**
    - To enable sending messages only after the websocket connection is established.
    - To control the visibility of the AlertsComponent.
    - To manage the state of the textarea input.

- **Custom authentication**
    - Why is `request.FILES` important when processing the UserProfileForm?
        - **It contains the data from the POST request.**

    - It contains the uploaded files.
    - It handles user authentication.
    - It defines the form's appearance.

- What is the primary reason Django Admin's authentication module cannot be used for all users?
    - It is too complex to configure
    - It requires a paid subscription
    - **It is limited to administrator users**
    - It is incompatible with other Django modules

- Where should you place the code that exposes the URLs for serving uploaded files?
    - **In the application's `urls.py` file.**
    - In the project's `urls.py` file.
    - In the `account/urls.py` file.
    - In the `settings.py` file.

- What is the recommended approach to handle file uploads, specifically for the user avatar?
    - **Directly modifying the User model.**
    - Creating a one-to-one relationship with a separate model.
    - Using a built-in Django file upload widget.
    - Ignoring file uploads altogether.

- What is the benefit of using a global configuration like MEDIA_ROOT for media files?
    - **It allows you to easily modify the root folder for all media type files in one place.**
    - It automatically creates a separate folder for each image type field.
    - It prevents the need to validate image sizes.
    - It eliminates the need to delete previous avatar images.

- According to the text, what is the primary container element used for the form in the login template?
    - Div
    - **Card**
    - Form
    - Alert

- What is the purpose of the `MEDIA_URL` setting in Django?
    - **It defines the base URL to serve multimedia files.**
    - It specifies the physical location of media files on the server.
    - It sets the maximum file size for uploads.
    - It determines the default language for the application.

- **Introduction to Testing**
    - What is the primary purpose of the 'setUp()' method in test classes?
        - **To define the main test logic for a class.**

    - To initialize the Client or create common data needed across multiple tests.
    - To clean up data after each test.
    - To import necessary modules.

- According to the text, what is a fundamental tool for ensuring an application works correctly?
    - **Tests**
    - AI interaction
    - Django ORM
    - RequestFactory

- What is the recommended way to run a test that utilizes a TestCase class, specifying the class and a method?
    - python manage.py test
    - python manage.py test comments.tests
    - **python manage.py test comments.tests.ContactTests.test_post**
    - python manage.py shell

- According to the text, what is the purpose of using a volatile, in-memory database during testing?
    - To simulate a production environment
    - To ensure data persistence between tests
    - **To provide a clean and independent environment for each test**

    - To speed up database operations

- What is a crucial limitation to consider when a test returns a 200 status code?
    - The application is guaranteed to be functioning correctly.
    - The test only verifies the presence of specific HTML elements.
    - **A 200 status code doesn't necessarily mean the application is working correctly; the content might be faulty.**
    - The database connection is automatically established.

- What status code does the view return after a successful POST request, as described in the text?
    - 200
    - 301
    - **302**
    - 404

- Why are API tests often separated into their own files and use `APITestCase` and `APIClient`?
    - Because the API tests require more complex setup.
    - **To optimize message handling and simulation of REST API requests using JSON.**
    - To simplify the codebase by separating concerns.
    - Because Django Rest Framework requires it.

- **General Aspects**

- What does the `.delay()` method do in the context of Celery?
    - **It executes the task immediately on the FastAPI server.**
    - It serializes the task and sends it to Redis.
    - It defines the Celery configuration parameters.
    - It checks the status of a task in the Celery backend.

- What is the primary purpose of using `django.core.signing.dumps()`?
    - To encrypt data for secure storage
    - **To sign and encode data for retrieval**
    - To hash data irreversibly
    - To generate a digital signature for authentication

- What is the main difference between using `request.session` and Django's Messages Framework?
    - `request.session` is ephemeral, while Messages Framework persists longer
    - **`request.session` is for storing user-specific data, while Messages Framework displays temporary notifications**
    - Messages Framework uses the `SECRET_KEY`, while `request.session` does not
    - There is no difference; they are interchangeable

- According to the text, what is the primary purpose of Celery compared to FastAPI's native BackgroundTasks?

    - **Celery is designed for production environments where reliability and scalability are priorities.**
    - Celery is simpler to implement for basic background tasks.
    - Celery's BackgroundTasks are not lost if the server restarts.
    - Celery is only suitable for simple tasks like sending emails.

- What is the purpose of throttling in Django REST Framework?
    - To improve performance
    - **To protect against resource abuse**
    - To simplify API authentication
    - To automatically generate documentation

- According to the text, what is the role of the `SECRET_KEY` in Django's cryptography?
    - It's used for compressing data.
    - **It's the key used to generate signatures and protect sensitive data.**
    - It's used for hashing passwords.
    - It's the key used for encrypting data.

- What is the primary purpose of Django's `Signer` class?
    - To encrypt data for secure storage.
    - **To generate digital signatures to validate data integrity.**
    - To hash passwords for secure authentication.
    - To compress data for efficient transmission.

- **Essential Packages**
    - What is the main function of Django Rest Framework?
        - To customize the default appearance of Django forms.
        - To enable CORS headers for cross-origin requests.
        - **To build a complete REST API using Django.**
        - To manage persistent connections for real-time communication.
    - Why is Django Unfold considered the most modern and recommended option for customizing the Django Admin?
        - It's based on Bootstrap 4.
        - It's a classic package with a modern aesthetic.
        - **It's based on Tailwind CSS and has active maintenance.**
        - It's the easiest to install and configure.
    - According to the text, what is the primary purpose of the Django Debug Toolbar?
        - **To check the status and efficiency of your application during development.**
        - To build a complete Rest API using Django.
        - To enable CORS headers for cross-origin requests.
        - To customize the appearance of Django forms with Bootstrap 5.
    - What is the primary reason Django-Seed might encounter compatibility issues, according to the text?
        - **The package hasn't been updated to Django 4.**

        - Complex relationships in models.
        - Issues with random data generation.
        - Problems with external files.

- What is the purpose of the .env file in Django environment variable configuration?
    - **To store project-specific configurations without affecting the codebase.**
    - To define the Django models.
    - To install Django packages.
    - To manage database connections.

- What is the primary purpose of the Django Debug Toolbar?
    - To automatically generate image thumbnails.
    - To provide a modern aesthetic for the Django Admin.
    - **To display detailed information about the application's operation and database queries.**
    - To simplify the installation and configuration of third-party packages.

- What is Django Channels primarily used for?
    - Customizing the appearance of Django forms.
    - **Managing persistent connections for real-time communication.**
    - Importing and exporting data in your Django application.
    - Generating test data for Django models.

- **Deployment**

- According to the text, what type of servers are typically used for Python web applications instead of those used for PHP?
    - Shared servers
    - **VPS (Virtual Private Server)**
    - Dedicated servers
    - Cloud servers

- When deploying to PythonAnywhere, what should you do with the original WSGI file?
    - **Delete it entirely**
    - Copy it to a separate file (e.g., PythonAnywhereoriginal.py)
    - Modify it directly without making a backup
    - Rename it to wsgi_modified.py

- What file must be included at the root of a Python project (including Django, Flask, and FastAPI) for Railway to automatically install dependencies?
    - Dockerfile
    - README.md
    - **requirements.txt**
    - setup.py

- What is the purpose of the environment variable `ENV PYTHONDONTWRITEBYTECODE=1`?
    - **Prevent Python from generating .pyc files.**

    - Automatically generate .pyc files.
    - Enable bytecode caching.
    - Disable Python interpreter.

- What is the purpose of `host.docker.internal`?
    - To connect a container to the internet.
    - To connect a container to a different Docker network.
    - **To allow containers to communicate with the host machine's services.**
    - To automatically deploy applications to the host.

- What is the primary purpose of Railway, as described in the text?
    - To host PHP applications
    - **To simplify cloud deployment for developers**
    - To manage MySQL databases
    - To provide a free hosting service for all programming languages

- According to the text, what is the recommended option for a Python web server in a production setting?
    - **gunicorn**
    - waitress
    - uWSGI
    - nginx

Preview

This is a Preview of the book content and it is completely FREE, you can get the full book from: **desarrollolibre.net/libros**.

Chapter 1: Required software and installation of tools

Python is an excellent language to start programming, it is simple, modular, and due to its very structure, when using indentation to delimit the structure, the code is correctly indented; with basic knowledge of Python, we can use them to master much of the framework as we are going to learn in this book.

Required Software

As fundamental software, we need Python, this is the only program that you have to install on your computer, since it is the programming language in which Django is programmed, and the one that we have to use to create our applications with this framework; the Python website is this:

https://www.python.org/

Python

Before we go any further, let's talk a bit more about Python; Python is a high-level and interpreted programming language, object-oriented and with a very clean, friendly syntax, easy to learn and great readability with a large number of packages, modules, libraries and frameworks at our disposal make it very attractive for rapid application development.

> *What is it? Python is the programming language that supports Django. A framework (like Django) is simply a framework or a set of reusable tools, but it needs a base programming language.*

Programmers often fall in love with Python because of the increased productivity it provides. Since there is no build step, the cycle of editing, testing, and debugging is incredibly fast and easy.

Before installing, we will check if there is already a version of Python installed on the system.

For some operating systems, such as Linux and MacOS, they come with a version of Python installed by default; to determine this, just run in your terminal:

```
$ python -V
```

Or

```
$ python3 -V
```

In my particular case on my Mac, the response from the terminal is:

```
Python 3.12.0
```

Respectively; this indicates that I have version 3.x installed.

It is important to note that you may be able to run by simply saying python or python3; it all depends on what you have installed on your computer; normally on both Linux and Mac, they come by default with both versions of Python; but you may also be able to run Python version 3 by simply typing "Python" in the terminal; therefore, the command that your Python version 3 returns is the one you should use to develop in Django, since recent versions of Django work with Python version 3.

Finally, if the version that appears in the terminal is supported by the minimum version required by Django:

> *Django versions Python versions*
>
> *2.2 3.5, 3.6, 3.7, 3.8*
>
> *3.0 3.6, 3.7, 3.8, 3.9*
>
> *3.1 3.6, 3.7, 3.8, 3.9*
>
> *3.2 3.6, 3.7, 3.8, 3.9, 3.10*
>
> *4.0 3.8, 3.9, 3.10*
>
> *4.1 3.8, 3.9, 3.10, 3.11 (added in 4.1.3)*
>
> *4.2 3.8, 3.9, 3.10, 3.11, 3.12 (added in 4.2.8)*
>
> *5.0 3.10, 3.11, 3.12*
>
> *6.0 3.12 +*

https://docs.djangoproject.com/en/dev/faq/install/

There is no need for you to install a new version of Python since you can work with the one that already comes with your system.

Pip

Pip is simply a tool that is installed along with Python, and its purpose is to install Python dependencies or packages. When creating projects in Django, Flask, or FastAPI, we use this tool to install the corresponding package.

Furthermore, it allows for complete management (listing, updating, and uninstalling) of these packages; therefore, it is a crucial tool for Python development in general.

You must follow the same procedure with the package installer for Python, called pip:

```
$ pip3 -V
```

Or

```
$ pip -V
```

In at least one case you should see something like:

pip 22.X from /Library/Frameworks/Python.framework/Versions/***/pip (python 3.X)

Basic commands

In order to create our applications in Python in an organized way, you must at least know the following commands, which are very useful when developing; let's see what they are.

To install dependencies in Python, which can be from a plugin to perform social login or a framework like Flask, FastApi or in this case Django, you must use the "pip install" option followed by the package or packages that we want to install; for example:

```
$ pip install flask
```

To uninstall a package, we have the "pip uninstall" command

```
$ pip uninstall flask
```

Another very useful command is the one that allows us to know which packages we have installed in a project; As happens with other environments such as Node or PHP with Composer, in Python many times when the installed packages contain dependencies; for example, in the case of Django, when you install Django, the following dependencies are installed:

```
$ pip install django
asgiref==3.7.2
Django==5.0
sqlparse==0.4.4
tzdata==2023.3
```

To know the dependencies, we have the following command:

```
$ pip freeze
```

Whose output we can save in a file, which by convention is called requirements.txt:

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

We can now use this file to install the dependencies of a project with the exact versions; very useful when you are developing a Python project with multiple people or when you move to production or simply when you need to run the project on multiple PCs:

```
$ pip install -r requirements.txt
```

But, in case you need to install Python, it's extremely easy; we go to the official website, in the download section:

https://www.python.org/

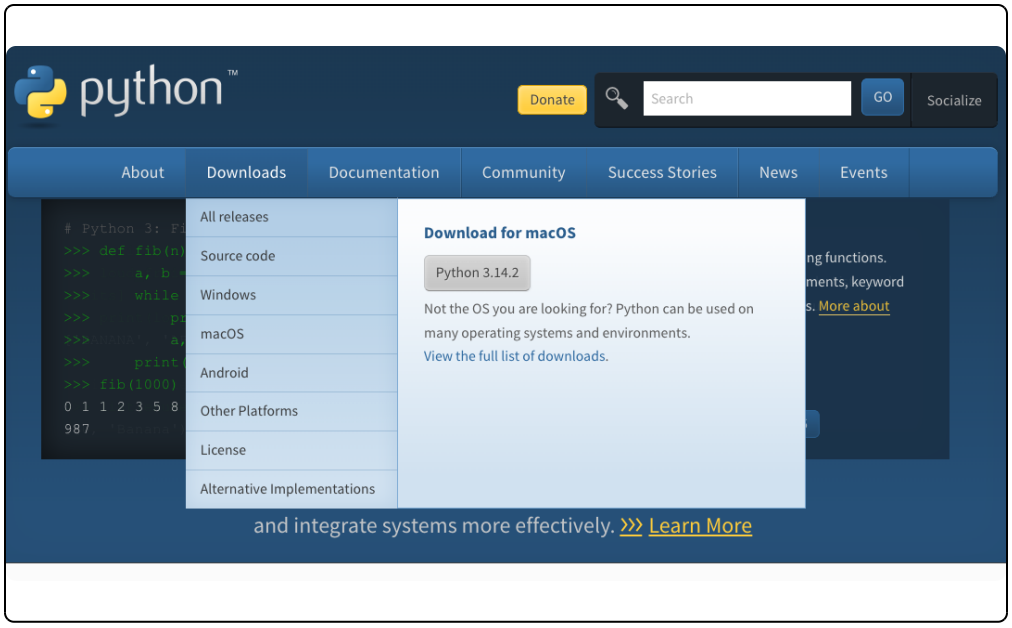


Figure 1-1: Python Web

We download it, run it and it is the typical "next next next" to install it; very important that you add Python to the Windows PATH:

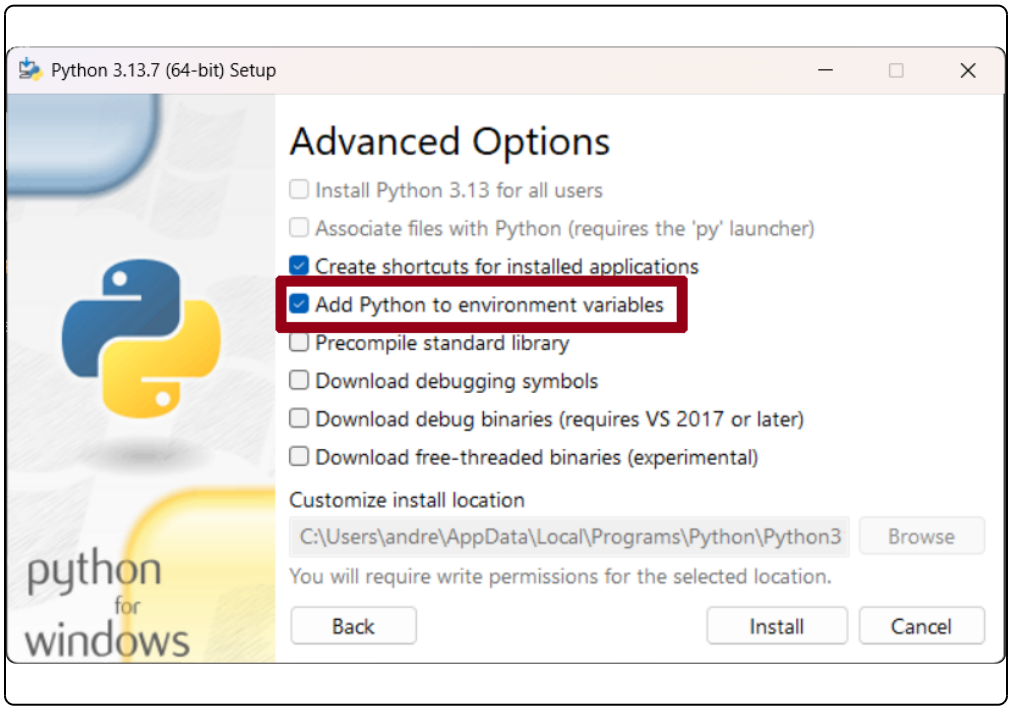


Figure 1-2: Add Python to the Path

> *Critical Installation Point on Windows*
> *When installing Python (running the .exe), remember to enable the option to add Python to your environment variables (Add Python to PATH). This is crucial, because if you don't add it, it becomes "orphaned" and you have to access it using the full path. If you forget, simply reinstall the .exe, and it will give you the option to enter repair mode and add the checkbox.*

Visual Studio Code

As a code editor, we are going to use Visual Studio Code since it is an excellent editor, with many customization options, extensions, intuitive, lightweight and that you can develop in a lot of platforms, technologies, frameworks and programming languages; so all in all Visual Studio Code will be a great companion for you; but, if you prefer other editors like Sublime Text, or similar, you can use it without any problem.

https://code.visualstudio.com/

Extensions

If you use VSC, you need to install the Python extension.

- Python (Microsoft): You absolutely must install this, otherwise, programming is like programming in Notepad. This is because internally, when we create our Django project (which is Python), navigation, recommendations, package imports, and error detection won't work without it.
    - https://marketplace.visualstudio.com/items?itemName=ms-python.python

- And as a recommendation, the following extension. Django: You can also look for extensions for Django (for example, for the Django Template) that help you with tags and more.
    - https://marketplace.visualstudio.com/items?itemName=bigonesystems.django

Browser

As a web browser, I recommend Google Chrome; although it is true that when developing web technologies, it is advisable to use more than one browser; by developing specifically for the server side and not focusing on client-side development; it doesn't make much sense for this book that you use multiple browsers; that said, you can use any other browser in case Google Chrome is not to your liking.

https://www.google.com/intl/en/chrome/

Database: MySQL

For the database, you don't need to install a database management system, as you can use SQLite perfectly well. However, in some tests, we will optionally use MySQL. In those cases, there are many ways to install it depending on your operating system. For Windows and macOS, you can use Dbngin to install MySQL:

https://dbngin.com/

And as a handler, I recommend Tableplus:

https://tableplus.com/

Questions

- According to the text, what is the only program you absolutely need to install on your computer to start programming with Django?
- What is the recommended browser for development, as mentioned in the text?
- What is the purpose of 'pip' in Python development?

You can practice the exercises interactively in: **Academy**

Chapter 2: Getting started with Django

With our ecosystem ready, which in a few words is our Python, and our code editor, now we are going to install our Django; but, before this, we are going to cover a couple of important points.

Packages and best practices when working with Python

Surely if you have consulted a tutorial on how to take the first steps in Python or an associated framework, you will have noticed that they generally use a virtual environment and this is so, since, with it, in a few words it allows you to install Django, and its dependencies, and any other packages you want to install in isolation from the operating system; this is so, since if you want to install Flask, Tkinter, Django or any other, without using a virtual environment, all of them are installed at the operating system level.

The "problem" you have when working with Python packages, by which we understand any package, is that they are installed at the global level of the operating system, and with this, if, for example, you want to work on a project with Django 4, but you have another project with Django 3, you will have to uninstall and install the specific version of the framework in each of these projects; that is, if you are going to work with Django 3, you must have the Django 3 dependency installed, and the same with

Django 4.

> ***When you install Django**, you'll see that it has a lot of dependencies. That is, **the installation includes not only Django** but also other related packages. If you've worked a bit with the Python ecosystem, you know what I mean.*
>
> ***For example, when installing Django 5, you'll have three or four additional packages**. This happens because Django, unlike micro-frameworks (like Flask), comes with many built-in features (templates, authentication, etc.). Flask, being a micro-framework (which includes the bare minimum), didn't even initially include the template manager (Jinja2), and you had to install it as an additional dependency.*
>
> ***The main idea is this: you install Django, and three or four more packages are installed.***
>
> *The problem is that, in the real world, you're going to have many projects.*
>
> *If we were to install Django globally (as with Python, a browser, or VS Code), you could **only have one version of Django installed on your operating system** unless you made complex configurations.*

This to give an example, but the same will happen when working with any other technology, you must keep in mind that, when installing a package, it may have specific dependencies; for example:

Django 6 project:

```
asgiref==3.11.0
Django==6.0
sqlparse==0.5.4
tzdata==2025.2
```

Django 5 project:

```
asgiref==3.7.2
Django==5.0
sqlparse==0.4.4
tzdata==2023.3
```

Django 4 project:

```
asgiref==3.5.0
Django==4.0.3
sqlparse==0.4.2
tzdata==2021.5
```

Django 3 project:

```
asgiref==3.4.1
Django==3.2.8
sqlparse==0.4.2
```

So, in a nutshell, to be able to develop on any technology that uses Python, it is highly recommended that you use a virtual space.

Virtual environments

To solve this situation, there are virtual environments or environments, which, in a nutshell, allow you to create a system isolated from the operating system and with this, we can install the dependencies in this virtual environment:



Figure 2-1: Virtual environments in your operating system

With this, in the example above, we have:

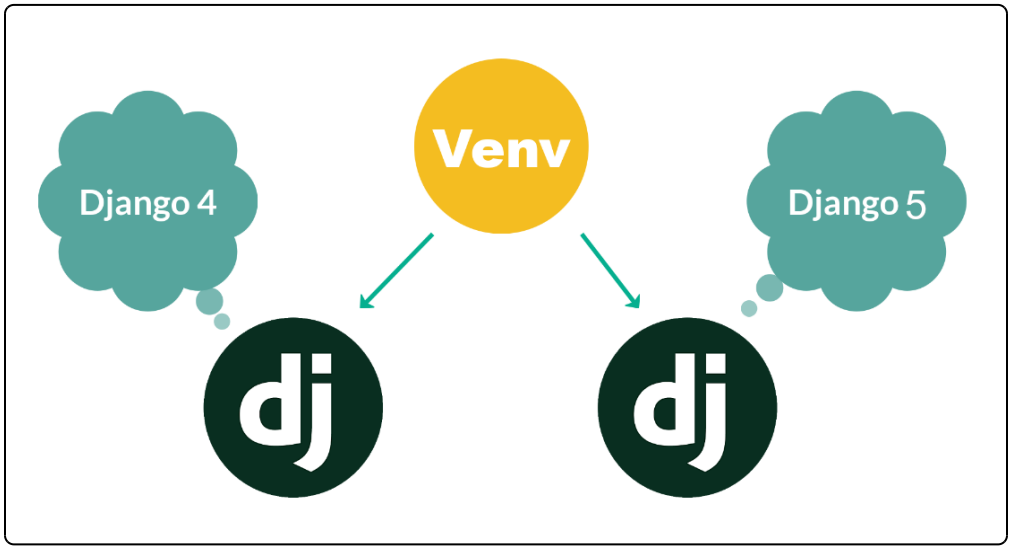


Figure 2-2: Virtual environment for projects in Django

In summary, what do virtual environments allow us to do?

Instead of installing the packages globally to the operating system, a kind of "box" (a virtual environment) is created and everything is installed right there.

- Global Operating System: If we only use the operating system, we can only have one installation of a version.
- Virtual Environments: If we use virtual environments, we can have "n" virtual environments for all our projects.

That is, we can have the environment that the Django 5 project needs installed and also the Django 6 project, both in different virtual spaces; and so on, for any other type of project regardless of whether it's related to Django or another framework like Flask.

Prepare the development environment

In this section, we are going to prepare the virtual environment, which is necessary to be able to develop effectively in any technology that has to do with Python and in Django, it cannot be the exception.

Install virtual environment

There are some virtual environment for Python:

- pyvenv
- venv
- conda

To name a few; We are going to use the so-called venv since it is the most used today and even the one recommended by the official Python documentation:

https://docs.python.org/3/library/venv.html#module-venv

This package **is the only one that we will need to install globally** throughout the book; which allows you to create virtual spaces; to install it:

```
$ pip install virtualenv
```

Create virtual space manually

With the previous package installed, we're now going to create a virtual space; first, I'll show you how to do it manually, but I recommend using VSC, which we'll cover in the next section:

> ***Location**: You need to navigate to your project folder. For example, I have a folder on my Desktop called "projects," and inside it, I created a folder called "test" (since I won't be using this virtual environment).*
>
> ***Accessing the Folder**: Open your terminal (CMD) and type cd (Change Directory), then drag the folder to the terminal (Windows, Linux, and Mac support this) or navigate to it manually.*

```
$ python -m venv firststeps-env
```

- **python** -m: Indicates that a Python module will be executed.
- **venv**: This is the module you installed to create virtual environments.
- **environment_name**: The name you choose. (I usually use an underscore, like test_venv, to identify it as a virtual environment.)

This will create a folder with the name you define for your virtual space; in this book it is "firststeps-env"; access with cd:

```
$ cd firststeps-env
```

You can review the generated folder, its organization and the files and you will see that a virtual space is just that, a set of folders and files that we can activate on demand and with this, it creates the isolated space of our operating system in which we can create our projects in Python.

An important point is to note that generally a project is tied to a virtual space, and vice versa; which means that we use a virtual space exclusively to work on a project, and this is so to avoid installing dependencies of two or more projects in the same virtual space; in Python it is important to know which packages our project requires and which versions, something that we could not clearly know if we have several dependencies installed from different projects, in the same virtual space.

How do you know if you already have it installed? Simply enter the command:

```
$ pip freeze
```

And you'll see the packages installed globally on your system. I, for example, have installed: virtualenv and pip (and their dependencies):

```
distlib==0.3.9
filelock==3.18.0
platformdirs==4.3.8
virtualenv==20.31.2
```

Activate virtual space

Already with the virtual space created, now we must know how to use it; we can activate the virtual space that we created earlier.

On Windows with CMD:

```
$ .\Scripts\activate
```

On Mac and/or Linux:

```
$ source bin/activate
```

> *To activate it, we need to navigate to the Scripts subfolder (on Windows) or bin (on Linux/Mac) within our virtual environment. The command varies depending on the operating system.*

In any scenario, once activated, something should appear in the terminal, the name of the virtual space:

```
(firststeps-env-env) C:\Users\Desktop\firststeps-env>
```

Extra: Practical Verification

Now comes the test to help you understand how virtual environments work:

- **Inactive Terminal (Global**): If you type pip freeze in a new terminal or command prompt, you'll see the packages installed globally on the operating system (virtualenv and its dependencies).
- **Active Terminal (Virtual)**: If you type pip freeze in the terminal you just activated, you'll see that absolutely nothing appears.

This indicates that we now have a clean space to install Django or whatever else we need.

Deactivate virtual space

To deactivate it or turn off the virtual space we have.

On Windows with CMD:

```
$ .\Scripts\deactivate
```

On Mac and/or Linux:

```
$ source bin/deactivate
```

In any scenario, once active, the name of the virtual space that was in parentheses should disappear:

```
C:\Users\Desktop\first steps-env>
```

It is important to note that, when we are going to program in Django, you must always have it active; there is also no need to stop the process manually when you finish working (or when you turn off your computer).

Another important point is that, note that in Windows and in Mac/Linux we have a different structure and the command changes slightly.

Also, you can have as many active virtual spaces as you want, but try that for the same virtual space, you only have one open instance.

Optional VS Code: Creating the Virtual Environment using VS Code

This is the option I recommend. It creates the virtual environment directly from a VS Code instance.

- **Open Folder**: Open a clean instance of VS Code (Ctrl + Shift + N). Drag or open your project folder (the one I named first_step).
- **Create Environment**: Press Ctrl + Shift + P.
- **Search for: Python**: Create Environment.
- **Select the provider**: Venv (or Virtualenv).
- **Select your Python**: The Python version you have installed at the operating system level.

VS Code will do everything for you. The great advantage is that you don't have to deal with selecting the interpreter (the manual method) or with issues where it's sometimes recognized and sometimes not.

Once installed, open the terminal in VS Code. You'll see the name of the virtual environment in parentheses, indicating that it's active. If you run pip freeze again, you'll see that absolutely nothing appears.

Optional VSC: Select Interpreter

Most of the time VSC will not recognize the Python interpreter that is configured when installing a virtual environment, otherwise it will take (the one installed at the global level of the OS) the system one, this brings the disadvantage that these warnings appear in the application imports:

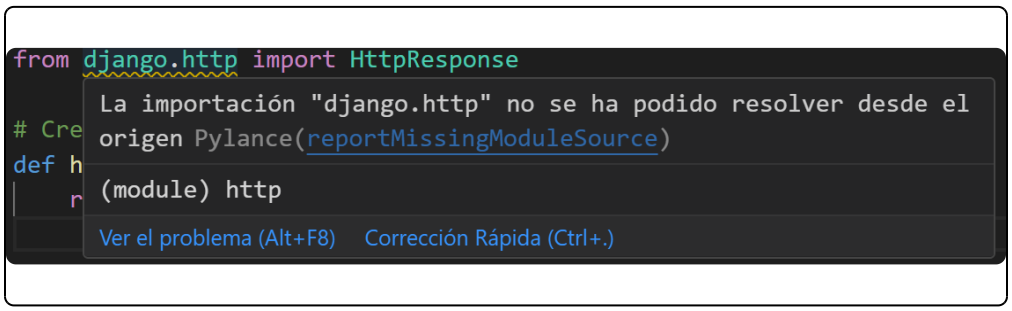


Figure 2-3: Select interpreter

And therefore, it is not possible to access the details of the imported modules; to solve this, we must tell VSC that the interpreter it should use is that of the virtual environment, which is where we have installed the entire ecosystem of the Django project and not that of the Operating System.

Press the key combination: Command/Control + Shift + P and search for "Select interpreter":

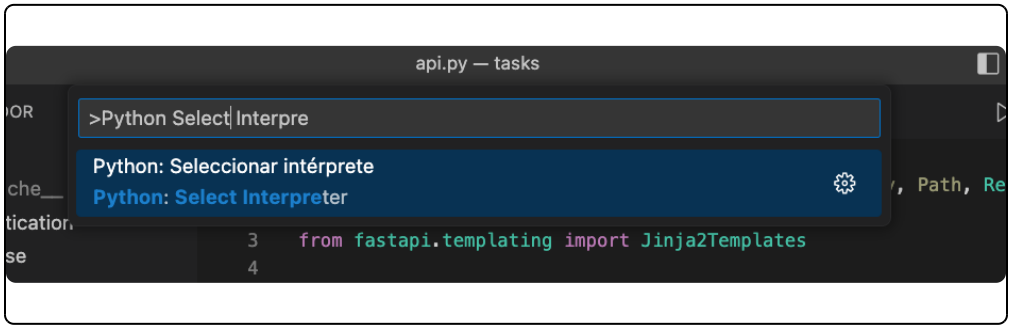


Figure 2-4: Window to select interpreter with Command/Control + Shift + P

And select your virtual environment from the option "+ Enter the path of the interpreter":

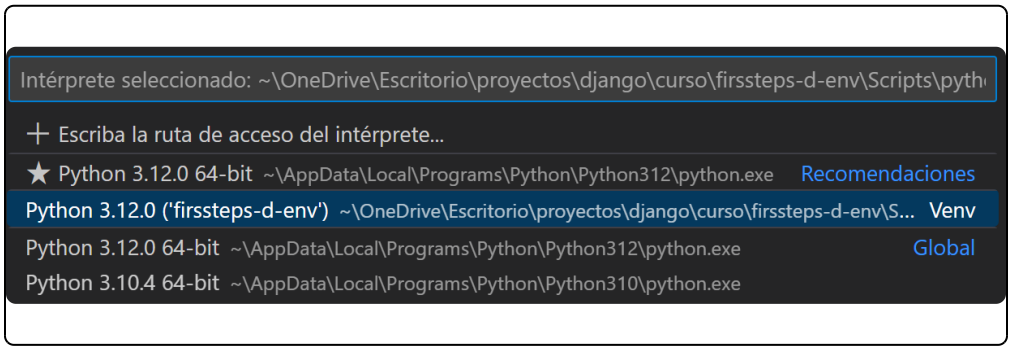


Figure 2-5: Select Python for the interpreter

In case, after selecting the interpreter, it still shows the warning:

*****Import could not be resolved "X"**

You can try creating a project from scratch using VSC; to do this, open a clean instance of VSC, then Command/Control + Shift + P and search for "Create environment":

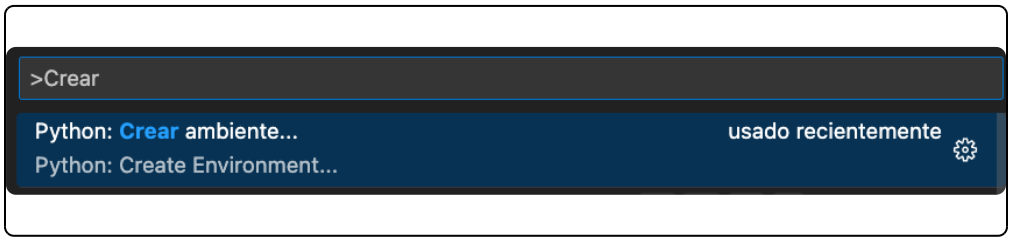


Figure 2-6: Create virtual environment

You follow the steps indicated by the editor, which would be to select the environment, place "Venv":

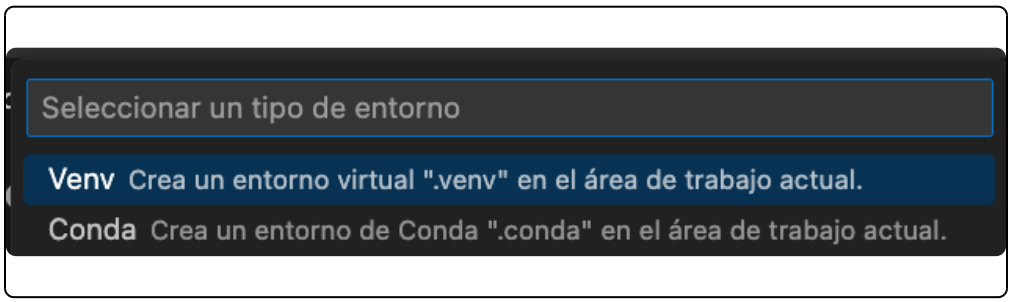


Figure 2-7: Select venv

Then, it may prompt you to place the work area:

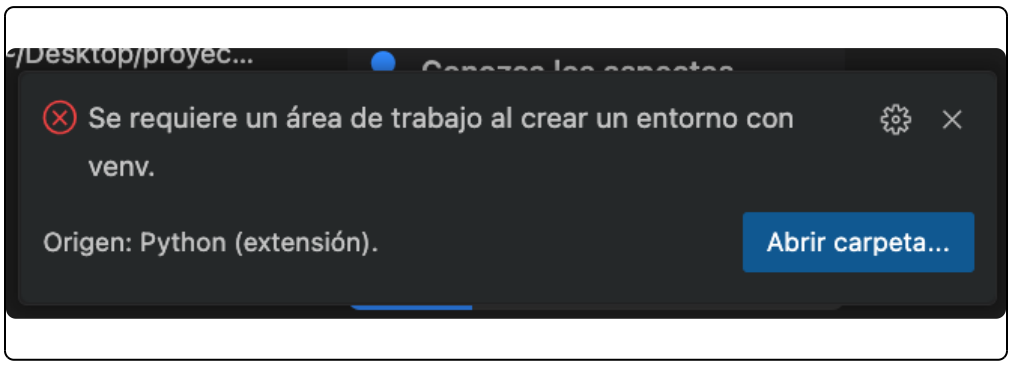


Figure 2-8: Select work area

And you go back to step 1, which would be Command/Control + Shift + P and search for "Create environment"; On this occasion, it should ask you for the interpreter you want to use to create the work area:

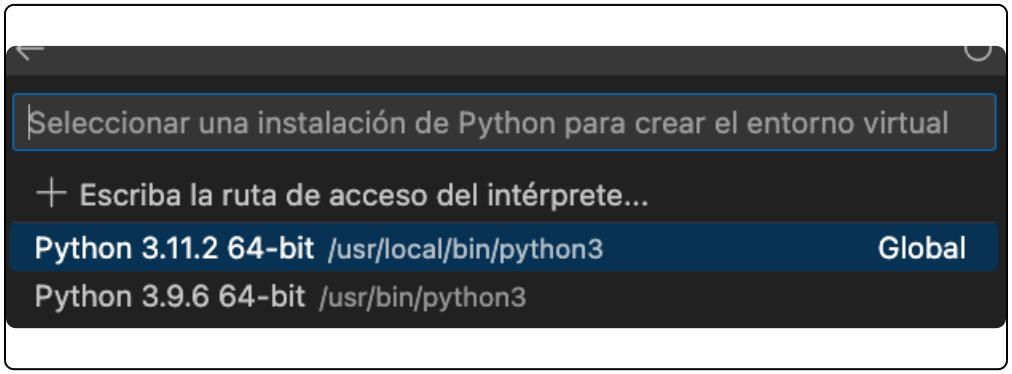


Figure 2-9: Select Python interpreter

And with this, the packages installed at the virtual environment level should be recognized.

Django

Now with all the necessary tools ready, let's work with Django, but before that, let's talk a little bit about the framework.

Before continuing, I want to quickly talk to you about why Django and why you should be doing this training.

Basically, I want to talk to you about this technology, which, for me, is my second favorite framework and the first when it comes to Python Web. (My first is Laravel, for all its fascinating features, but Django is not far behind and has many interesting points).

The main feature of Django is that it employs the Python ecosystem, which is excellent. Django is a web framework that we have at our disposal and has very interesting features.

Key Features of Django

As you can see here on the official page, Django is "ridiculously fast." It's a framework designed for developing applications quickly, and this is because it follows the "batteries included" philosophy.

This means it brings everything necessary for us to develop our applications professionally and as quickly as possible, something similar to what happens with Laravel. Unlike micro-frameworks like Flask or FastAPI, where the application is at its minimum expression and we scale little by little, Django already comes with the essentials.

Although everything has its advantages and disadvantages, these are the most important features:

- **Fast and Versatile**: It is a quite fast and very versatile framework, again, thanks to it being a Python project.
- **Asynchronous Support**: Since version 3 (if my memory serves me right), they added support for asynchronous functions, which is extremely useful for certain developments.
- **Included Components**: It has features already included, especially Django Admin, which we will learn about later. I think it is an excellent internal application because we can have automated CRUDs and implement our own in a very simple way.
- **Scalable and Modular**: It is scalable and modular (this is what I like most about this type of technology). Everything is based on components, and we can reuse pieces very easily.

The "Batteries Included" Concept

Here we have the "batteries included" term that I mentioned before. Automatically, with Django, we already have:

- **Django Admin**: The administration panel that I mentioned.
- **Authentication**: A ready-to-use authentication system (sessions, groups, etc.).
- **ORM**: The Object-Relational Mapper (ORM), which is the part for communicating with the database.
- **Migration System:** For generating the database tables.
- **Security**: Includes security features by default, which we will learn more about in detail when working with forms to avoid common problems.

Support and Non-Disruption

Django is scalable and is always being maintained. A major version usually comes out every one or two years (for example, two years passed from version 5 to 6, which is the current one).

The good thing is that it is not disruptive. That is, the changes are simply of form, new features are incorporated, and some packages are deprecated (obsolete) for security or efficiency reasons, but it does not break our code.

Basically, the same thing we can do in Django 4, we can do in Django 5 and Django 6, except for isolated exceptions. If you are using a feature that was modified, you will have to update it. That's why it's always a good idea to check the release notes (Release Notes or Checklist) on the official page to know exactly what the requirements and changes are.

Additional Scalability

One of the great advantages is its easy scalability. What is not defined by default (for example, a Rest API) can be easily installed.

An excellent package is Django Rest Framework, which we will see in the course and which allows us to create a Rest API with nothing to envy frameworks designed exclusively for that, such as FastAPI.

About Django

Django is a high-level, server-side web framework created in Python that promotes rapid development, a clean, functional, modular, and scalable design.

Being a web framework with basic features, we can expect to find components in it such as authentication, form handling and validation, an administrative panel, REST resources, route handling and linking, and many other functions.

Django's Modular Architecture (MVT)

Django is a great framework; it has a very active community and a lot of documentation. We will see it in a few moments when we start working and creating our famous CRUD.

For you to understand why it is modular, it works under the MVT (Model-View-Template) pattern.

Django implements a variation of the pattern known as MVC (Model, View, and Controller) in a peculiar way and with some variations that they call MTV, which is Model, Template, View.

But it has a practically one-to-one equivalence with MVC, in a change more of interception than anything else; the M, V, and C are separated in Django in the following way:

- **Model (M)**: The data control layer. It is the one that communicates with the ORM and uses the migration system to generate the tables. A model is a real-world object that we represent through a class (for example, if our application is about cars, we have a Car class).

- **View (V)**: The control layer (sometimes the most complicated to understand in its philosophical part). It is the one that communicates between the template and the model. All business logic (doing validations, sending an email, etc.) is handled by the view.
    - C, the layer that is responsible for connecting the Model with the View, this layer in Django could be said to be managed (or at least partially) by the framework itself; since it is Django that is responsible for managing the routing or handling of the routes (intercepting our client's request and sending it to the "Controller") which specifies which route is going to call which function in our project in Django URLConf, and for that reason, we have another type of layer that they simply call "View."

- **Template (T)**: The presentation part. This is where our CSS, JavaScript, and everything else goes. It is the HTML, but with Django's template engine, which allows us to add the dynamic part to the page.
    - It specifies what data we are going to display and defines the presentation of the data to our user. The only difference is that Django's views of our MVC are known as Template, therefore we go from V to T.

So in summary:

- **M** stands for **(Model)**, where it is still the layer responsible for communicating and interacting with the database.
- **T** stands for **(Template)**, in this case, the template fulfills the function of our view for the MVC, and it is what we use to display the data to our user, generally the HTML page.

- **V** stands for (**View**), in the MTV design pattern, it is the layer responsible for handling the business logic, through which the data will pass from the model to the template; this layer is the Controller in the MVC.

Obviously, we also have the routing part through the URLs.

In the following link, you can find the publication I made about this topic:

https://www.desarrollolibre.net/blog/python/el-patron-de-diseno-mtv-en-django-y-su-relacion-con-el-mvc

Install Django

Finally, we are going to install Django on our machine; with our virtual space active, we execute:

```
$ pip install Django
```

And this is it, we already have Django installed in our virtual space; but, we just install a package, and we don't have a project; but, with this package we can create projects in Django:

If we do a:

```
$ pip freeze
```

You'll see all the packages that make up the Django package we installed earlier; something like:

```
asgiref==3.11.0
Django==6.0
sqlparse==0.5.4
tzdata==2025.2
```

As we mentioned before, it is very common that when installing packages via the pip, these packages also need other packages to be able to work, and with this, the same ones are also installed, and these are known as dependencies.

In the example above, the asgiref, sqlparse, tzdata packages are dependencies of the Django package.

It is important to note that you will probably have some slightly newer versions and not exactly the same ones listed here (the version of each package is found after the ==).

As you can see, the previous command indicates all the packages installed at the virtual space level (if it is active) or at the operating system level (if the virtual space is deactivated or you are simply not using it).

Check the installed Django version

We'll assume you already have Django installed. You can find out if Django is installed and what version by running the following command:

```
$ python -m django --version
```

In our case it gives the following answer:

```
6.0
```

Which means we have version 6.0 of Django.

Projects and applications

In Django, we have to be clear about two key concepts, a project and an application.

A project in Django consists of one or many applications; that is, to be able to do something interesting in our project, we must have at least one application.

A project is a collection of settings and applications for a particular website. A project can contain multiple applications. An app can be in multiple projects.

An application in Django is a web application that does something, for example, a web registration system, a public records database, a Rest API, an app for selling something, etc.

In conclusion, the project contains the applications and configurations at a global level, such as the database, installed applications, middleware, templates, etc. and the applications are the ones that carry out the functions that we implement.

The use of Django to create applications is strongly linked to the modularization concept, and therefore, we can create multiple small applications that perform particular tasks, so that together, we can have the final application that we want; for example, for a blog type application:

1. An application for the list and detail of the posts.

2. An application to handle authentication, registration and derivatives.
3. An application to manage contacts, as well as manage post comments and others.
4. An application to handle a rest api.
5. An application to manage basic information, such as contact and "about".

The above list is an outline, and we didn't put the full description of what each app would do.

So, ultimately, the idea is to modularize our application (using the application concept as the final product); you don't have to follow this structure to the letter, you can add, remove or merge applications, but always keep in mind that the more modular the better the scalability and maintenance of the project.

Create a project in Django

To create a project in Django, we have the following command:

```
$ django-admin startproject mystore
```

This will finally generate the set of folders and files that make up a Django project that we are going to explain next.

It is important to note that "mystore" is the name of the project that we are going to create, but it can have any other name.

> *Notice that when we install the Django package, as I mentioned, we get a kind of CLI (Command Line Interface). This means we can use this special command.*
>
> *You can try this command with the virtual environment disabled, or directly in another terminal that doesn't have Django installed. If you run it, it will tell you: "What's wrong with you? This doesn't exist."*

Importante notar que "mystore" es el nombre del proyecto que vamos a crear pero, puede tener cualquier otro nombre.

Project structure

If you execute the previous command, you will see that a set of folders and files are created that we are going to explain.

These files are:

The root folder mystore/ (or whatever name you chose to create the project): This is just the container for the project; here are the settings and applications.

- manage.py: A command line utility that allows you to interact with this project.
- mystore/__init__.py: An empty file that tells Python that this directory should be considered a Python module. If you've worked with Python, you'll know that this file indicates that the folder containing it is a module. This reflects Python's philosophy of making everything modular, which is a major advantage of this ecosystem.
- mystore/settings.py: Settings or configuration for this project, among the main ones we have:

- **ALLOWED_HOSTS**: Here we'll put the domain we'll use when we deploy our application to production.
- **INSTALLED_APPS**: Indicates the installed applications. Note that we already have several pre-installed Django components (again, the "batteries included" concept): **django.contrib.admin**: The beautiful Django Admin.
    - **django.contrib.admin**: The beautiful Django Admin.
    - **django.contrib.auth**: For authentication (we already have a basic user model ready).
    - **django.contrib.contenttypes**, sessions, messages, and staticfiles.
- **MIDDLEWARE**: These are the intermediaries. We'll explain this later, but in short, when a user makes a request to our server, the middleware can be a class or function that runs in between to perform a task (for example, verifying access to an admin module).
- **TEMPLATES**: This is the template manager (Django's template engine) that allows us to manage the presentation.
- **DATABASES**: By default, we use SQLite as the database. (The database will be generated when we run the migrations later).

- **mystore/urls.py**: The URL declarations for the project.
    - For best practices, this file should only contain the URLs of the applications we're going to use (global paths), not the local paths of those applications. We create the local paths directly within each application.
    - You can see an example of this, where the application is included:

```
admin:urlpatterns = [
path('admin/', admin.site.urls),]
```

- **urlpatterns**: This part is crucial. The urlpatterns list must have that name, because it's what Django will read internally to load those URLs.

- **mystore/asgi.py**: Configuration file to serve the project on an ASGI type server.
- **mystore/wsgi.py**: Configuration file to serve the project on a WSGI-type server.

Command line

We are going to talk about the typical or most used commands in Django; it is not necessary that you know 100% what each one of them works for, if not, that you know their existence and understand each one of them in a basic way.

We have access to create a super user for the application and change its password; superuser that is necessary to enter the dashboard or administrator module for the application:

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

Commands to build projects and apps using the Django wizard:

```
$ python manage.py startapp
$ python manage.py startproject
```

To create and run migrations respectively:

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

Commands to start tests:

```
$ python manage.py test
$ python manage.py testserver
```

A command to run the server:

```
$ python manage.py runserver
```

In summary, don't worry about memorizing them all, but here are the ones we'll use most often:

- **createsuperuser**: To create the superuser and access the Django Admin panel.
- **makemigrations**: To create migration files (changes to your models).
- **migrate**: To run migrations and apply the changes to the database.

- **shell** (or sh): An interactive console for testing. Unlike the basic Python console, this one comes pre-loaded with the Django project, allowing you to reference models and the application.
- **startapp**: To start a new application (which we'll do soon).
- **test**: To run application tests.

Project settings

settings.py file and let's take a look at the project settings.

There are many configurations that Django includes in this file, but these are only part of it; the most important; the full list at:

https://docs.djangoproject.com/en/dev/ref/settings/

- **DEBUG** is a boolean value that toggles the project's debug mode on and off. If set to True, Django will display detailed error pages when your application throws an exception not caught; when you pass to production this value must be False to avoid exposing data related to the project structure.
- **ALLOWED_HOSTS** is not applied while debug mode is on or when tests are running. Once we go live and set DEBUG to False, you need to add the domain in this configuration to allow serving the Django app.
- **INSTALLED_APPS** setting tells Django which apps are active for this site. By default Django includes the following applications:
- **django.contrib.admin**: an administration site
- **django.contrib.auth**: an authentication framework

- **django.contrib.contenttypes**: a framework for handling content types
- **django.contrib.sessions**: a session framework
- **django.contrib.messages**: a messaging framework
- **django.contrib.staticfiles**: a framework for managing static files files
- **MIDDLEWARE** is a list containing the middleware to run.
- **ROOT_URLCONF** indicates the Python module where the application's URL patterns are located.
- **DATABASES** is a dictionary that contains the configuration of all the databases to be used in the project. There should always be a default database.
- **LANGUAGE_CODE** defines the default language code for this Django site.

These are just some configurations, don't worry if you didn't understand all the configurations, since we will be touching these sections in the following chapters.

Create an application

It is important to note that, by default, we do not have any application created in the project; here you can see that there is no place where we can program something to be displayed on the screen, in other words, the application.

To create the same:

```
$ python manage.py startapp comments
```

- **python manage.py**: The command line for our project.

- **startapp**: The command to start an application.
- **comments**: The name I'm giving the application. (I'm calling it comments because it will be used for simple CRUD operations; for me, CRUD operations are the fundamental tool for understanding any technology at a basic level.)

And with this, you will see that there is now a new folder:

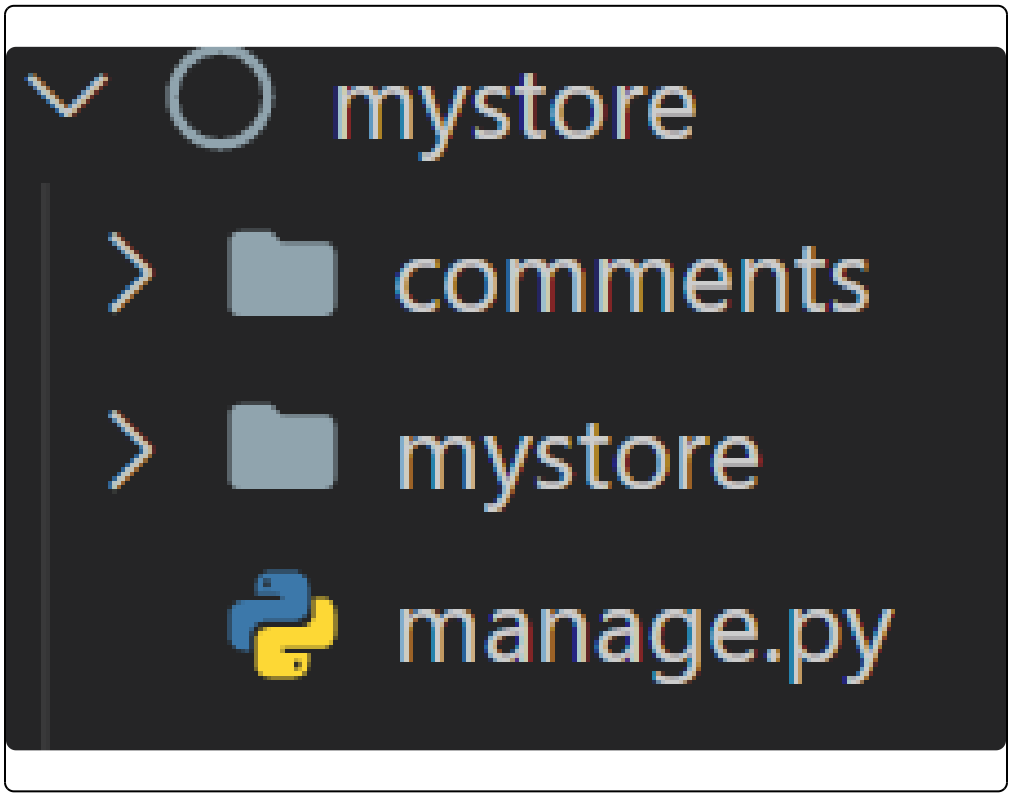


Figure 2-10: Project and application

Notice that part of the basic elements appear within the application:

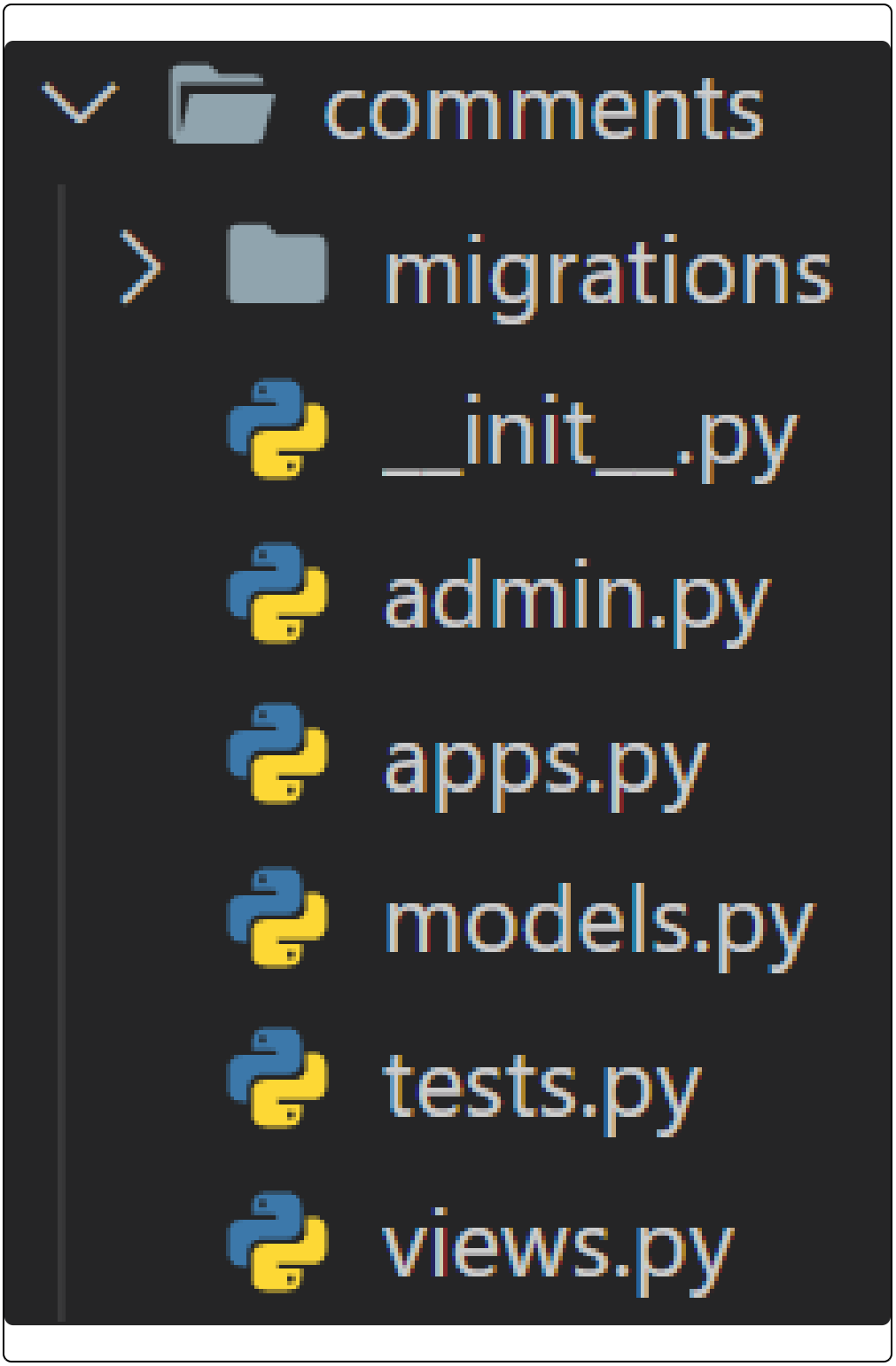


Figure 2-11: Application files and folders

Which, ultimately, are the elements that we are going to program.

> *Previously, we created the project using the **django-admin** command because the manage.py file didn't exist. Now that we have the project, we have this file available.*
>
> ***A project is like a parent** that contains a bunch of applications (the children or modules).*

Register application in the project

In order to use an application in Django, we have to register it; for this, at the project level, we modify the settings file:

settings.py:

```
INSTALLED_APPS = [
***
'django.contrib.staticfiles',
  'comments'
]
```

Just add the apps to the end of the installed apps variable.

Already at this point you can feel proud, we have prepared several elements necessary to develop in the correct way in Django, which go from the installation of Python, such as virtual spaces, to installing the Django package, generating the project and at least one application and understand its structure and some commands that we will be using.

> *The Importance of INSTALLED_APPS*
>
> *To make this clearer, let's restart the development server with runserver.*
>
> *Notice that we can access Django Admin by entering:*
>
> *http://127.0.0.1:8000/admin/login/*
>
> *This appears because the admin application is configured by default in the settings.py section, specifically within the INSTALLED_APPS subsection:*
>
> *mystore\mystore\settings.py*
>
> *INSTALLED_APPS = [*
> *'django.contrib.admin',*
>
> *Deactivation Test*
>
> *What happens if we remove it ('django.contrib.admin',) or comment it out (using Ctrl + KC or Command + KC in VS Code, or by adding the hash symbol #)? If we go back:*
>
> *http://127.0.0.1:8000/admin/login/*
>
> *Notice that it won't work anymore, just as you might expect.*

Already from the next chapter, we are going to start programming in Django.

Extra: Cloning a Django Project with Git and Github and Configuration

I'll show you a key process once the project is synchronized with GitHub, which you can use if, for example, you want to **work on different machines or with a team**. It's one of the great advantages of having the code hosted on GitHub.

1. Repository Cloning

The first step is to move to the directory where we want to copy the project and then clone the repository URL:

- Navigate to the Directory: Use cd to go to the destination folder.
- Clone the Repository: Copy the GitHub URL and execute:

```
$ git clone <URL_del_repositorio>
```

2. Local Environment Setup

Once cloned, you have the source code (the files), but you are missing the functional part: the virtual environment and the dependencies.

- **Autogenerated Files: Autogenerated folders (such as the virtual environment or vendor) should not be synchronized**. For example, on Windows it's called Scripts and on Mac/Linux it's called bin, so they are not directly compatible.
- **Renaming (Optional):** You can change the name of the cloned folder if you wish. I renamed it to mystore to maintain consistency.

A. Virtual Environment Creation

- Drag the folder to VS Code and create the virtual environment:
- Create Environment: Use the VS Code option to create a new virtual environment.

- Select Interpreter: Choose the Python version you want to use.

B. Dependency Installation

Although we only have Django right now, in a real project you will have many dependencies, and this is where requirements.txt shows its value since, with a single command, we install everything.

Isolation: This process highlights the importance of having an isolated virtual environment: each project has its own dependencies without mixing with others.

To install everything necessary, we use the file we generated before:

```
$ pip install -r requirements.txt
```

This automatically installs all dependencies listed in the file, without the need for manual commands.

3. Startup and Verification

Once the dependencies are installed, we start the server to verify that everything works:

```
$ python manage.py runserver
```

Questions

- What is the primary benefit of using a requirements.txt file when cloning a Django project?

- What is the main advantage of using venv over other virtual environment tools like conda?
- What command is used to create a new Django application within a project?
- What is the purpose of the INSTALLED_APPS variable in Django's settings.py?
- What command is used to activate a virtual environment on Windows using CMD?
- According to the text, what key combination should be pressed to search for "Select interpreter"?
- Why is using a virtual environment recommended when working with Python packages?

You can practice the exercises interactively in: **Academy**

Chapter 3: Introduction to Python

In this section, we are going to give an introduction to Python; you can take this section as reinforcement for the development of the framework that we will do in the next section.

It is important to note that this section is a reinforcement and not a introduction to programming, therefore, it is assumed that the reader has at least basic knowledge of programming and Python in general.

??Interactive console/Python console

The first tool that you can use when developing in Python is the interactive console, once Python is installed on your computer and the environment variable has been configured so that it can be used through the terminal, by opening a CMD or Terminal and writing:

```
$ python
```

The characters of:

```
>>>
```

Indicate that the console is waiting for you to type something, and they are not characters that you should type when performing the various tests that we will do in this section.

You will see that the console is active ready to carry out the first tests with Python, from here you can create variables, functions, lists, classes, etc., all for testing purposes, and this tool is ideal for when you want to learn about developing in the language or perform some tests with a script.

For example, if you type a number, you will see the output on the screen:

```
>>> 6
6
>>>
```

Or a text:

```
>>> hello world
hello world
>>>
```

In short, you can take the Python interactive console as another tool that you have easily available for testing.

Python files

The traditional way to develop in any programming language is through a file that is then compiled or interpreted, as in the case of Python, you can create a folder in any location on your computer that we will call:

```
C:\Users\andres\Escritorio\proyectos\python
```

From here we will develop the different scripts that we will see in this section.

We will create a file called:

test.py

In the case of this writing, we place the file in the previous location on the desktop and from here, we will do a hello world:

```
C:\Users> cd C:\Users\andres\Escritorio\proyectos\python
```

The **print()** function allows you to print a text through the terminal or CMD and is a tool that we will use greatly to quickly debug when developing the different functionalities of our application, so, following our example we put:

test.py

```
print("hello world")
```

And to execute, from the terminal and without running the interactive Python console, we execute:

```
$ python test.py
```

And we will see through the console:

```
hello world
```

You can also run the above content from the interactive Python console:

```
>>> print("hello world")
hello world
```

In this section, you can decide when to use one or the other depending on your preferences.

Variables and data type

The first thing we must know in any programming language is the use of variables to store data and data types, which are the primitives in any programming language. In Python, we can store numbers and text strings or strings; specifically:

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict

- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview
- None Type: NoneType

Python is not a language in which we must enter the data type when declaring the variable, it is dynamically typed, which means that Python infers it when initializing a value:

```
n = 5
```

Otherwise, as in JavaScript, the type is inferred based on the declared value; if it is an integer:

```
n = 5
```

...

Chapter 4: Create a CRUD application

The purpose of this chapter is to know the basic elements of the framework, such as the use of MTV, form handling, form validation and little else.

In this chapter we are starting from the structure that we created earlier; that is, having Python installed, our virtual space installed and configured, created a project and application called comments in Django.

Let's create another application with:

```
$ python manage.py startapp elements
```

For now, we will only use this application to define a series of models that we will need to delve a little deeper into the interoperability of applications, thanks to the modularization of Python projects in general.

We registered in the project:

settings.py

```
INSTALLED_APPS = [
  'elements',
]
```

Models

Models are the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you are storing (the business logic). Generally, each model maps to a single database table and is one of the layers of our MTV.

And, ultimately, they are the layer that allows access to the database and is the only entry to it; each application in Django can have zero or more models, all of which live inside a file called models.py that is defined in each application.

A model is nothing more than a class, where said class will be reflected in the database to a table; that is, a model class (meaning a class that is defined within the models.py files) will correspond to a table in the database.

It is important to point out that, to create the corresponding tables in the database, it is necessary to execute a command, but we will see that later.

? Database Connection

Based on the models, in the following lessons, we'll see how to create migrations. A migration is simply a process that allows us to translate what we have in the models into another file, and that other file

executes SQL internally in our database.

Remember that, by default, we're using SQLite (the completely clean database, whose configuration is in the DATABASES section of the settings.py file). Later, I'll show you how to configure, for example, PostgreSQL.

You can read a bit more about the theory here if you're interested. Without further ado, let's create the model, but we'll do that in the next lesson.

Create models

Models are part of applications in Django; within the file called models.py as its name indicates, we can define all the models that are going to be used in the application or other applications through imports.

Like any class, this one allows us to modularize our application. If it were a puppy application, you would create Dog, in the singular, since the model represents an individual entity in the database (according to best practice conventions).

The latter, you can see in the description of the models.py

```
# Create your models here.
```

Case study

Inside the application we created earlier, we are going to create the following classes:

elements/models.py

```
from django.db import models
# Create your models here.
class Category(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255)
    def __str__(self):
        return self.title
class Type(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255)
    def __str__(self):
        return self.title
class Element(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255,blank=True)
    description = models.TextField()
    price = models.DecimalField(max_digits=10,decimal_places=2, default=6.10) #
12345678.10
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    type = models.ForeignKey(Type, on_delete=models.CASCADE)
```

Explanation of the above code

Here you can already start to see the wonders of Python; clean, expressive syntax and surely you already understand at least in a basic way what we are defining.

The first thing we do is import Django's Model class:

```
from django.db import models
***
```

models.Model

Next, we create classes that inherit from the previous import:

```
class Category(models.Model):
  ***
class Type(models.Model):
  ***
class Element(models.Model):
  ***
```

Providing these classes with certain characteristics that we will see little by little, which in this very particular case would be to manage the models, including connections to the database and mapping of the classes to the database through migrations.

Then, each of the previous properties defined in said class, make up the structure of the entity that we want to create; in this case categories, types and elements.

Here in this elements/models.py file, we create three different models, two of them are simple and are for category and type and the third one is a bit more elaborate intended to be able to register posts; posts which can be a post, course or any other typing you want to use; the term "element" is used in the book to place a neutral name for each of these types or classifications.

Let's defrag the Element class; understanding its columns, you will understand that of the other two classes and similar definitions:

- With **models.CharField(max_length=255)** for the title property, we define a field of type text (**CharField**) which allows characters to be stored, with an attribute to limit the maximum size to 255 characters (**max_length**).
- With **models.SlugField(max_length=255,blank=True)** for the slug property, we define a field of type SlugField, which allows storing only letters, numbers and hyphens which is ideal for handling clean or slug URLs.
- With the blank we indicate that it can store blank values.
- With **models.DecimalField(max_digits=10,decimal_places=2, default=6.10)** for the price property, we indicate a field of type decimal of a maximum length of 10 digits and two decimal parts with a default value of 6.10.
- With **models.ForeignKey(<Relation>, on_delete=models.CASCADE)** we will create a foreign relation of type in cascade, with the category and type.
    - **models.CASCADE**: If you delete the category, the related item is also deleted (cascading delete).
    - **models.PROTECT** or **models.RESTRICT**: Prevents deletion.
    - **models.SET_NULL**: If you delete the category, the item's category field is set to **NULL** (provided the field allows null values).

- For **models.TextField()** for the description property, we define a field of type text; this is ideal for storing large amounts of text.
- For the date fields for created and updated that represent the dates of creation and update of records respectively, we indicate the DateTimeField type field that with the **auto_now_add** attribute we indicate that it can set a date when generating the object and with the auto_now we indicate what to create the object, put the current date.

The reason for this structure is due to a publication of any kind in the real world, if you want to create a publication, the basic elements are a title and content; generally we indicate a summary or description, and of course some control dates are always welcome; finally, having many posts of various types, we add classifications as indicated above.

In the end, it is important that you familiarize yourself with the types of columns and attributes to use the ones that best suit your needs:

https://docs.djangoproject.com/en/dev/ref/models/fields/#field-types

For now, we will only use this application to create the models, which initially we will only use to import from the application (comments) that we created in the previous chapter.

Remember to register this application in:

```
INSTALLED_APPS = [
    ***
    'django.contrib.staticfiles',
    'elements',
    'comments'
]
```

In order to use this application.

Inside the application that we created in chapter 2 we are going to create the following model class:

comments/models.py

```
from django.db import models
from elements.models import Element
# Create your models here.
class Comment(models.Model):
    text = models.TextField()
    date_posted = models.DateTimeField(auto_now_add=True)
    element = models.ForeignKey(Element, related_name='comments',
on_delete=models.CASCADE, null=True)
    def __str__(self):
        return 'Comentario #{}'.format(self.id)
```

Explanation of the above code

From this app, we are importing a model that is defined in the app called elements:

```
from elements.models import Element
```

To load a file from another module (in this case remember that a module is the application), we use the name of that module, followed by the file we want to load; example from **elements.models** import Element

Otherwise, it has a structure like the one explained above in the application models elements.

If you wanted to create a comment in the real world, you would obviously need to have the content of it (the attribute text), it would also be convenient to store the date (**date_posted**) and to whom it is directed (element); of course each of these features have to have text, date as it is addressed; thinking about that, we have the structure presented above.

Optional: Gemini CLI and/or Gemini Agent, Your AI Assistant

Before starting to develop, I want to make a quick recommendation so you have a "friend" to assist you.

I suggest you install or configure an Artificial Intelligence (AI) assistant. In this case, I recommend Gemini, which is completely free, although you can also use Claude or whichever one you prefer.

1. Gemini Code Assistant (VS Code Extension)

We're going to quickly review two tools. I will leave the detailed information (step-by-step) in a video on my YouTube channel and on my blog.

First, I recommend you search in the Extensions section of VS Code:

You will see the assistant that you can install. You should always watch out for it to say "by Google" and have the verified checkmark (it's usually the first one):

https://marketplace.visualstudio.com/items?itemName=Google.geminicodeassist

Then, a window will appear for you to authenticate (with a large Login button).

Once authenticated, you will be able to ask anything associated with this project directly in VS Code.

2. Gemini CLI (Command Line Interface Tool)

Another tool that I like even more is the Gemini CLI (Command Line Interface).

https://www.desarrollolibre.net/blog/ia/gemini-cli-step-by-step-installation-and-use-for-developers

🕶️ecommendation and Modern Programming

3. Error Identification and Correction

Let's suppose I intentionally made a mistake, for example, in the indentation of a field, and the name of a field:

```
class Type(models.Model):
    title = models.CharFILE(max_length=255)
slug = models.SlugField(max_length=255)

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

If you have doubts about your code, or simply want a check (even if the editor tells you the error), you can ask Gemini directly.

1. Selection and Query: You can select the block of text containing the error (for example, the Element model) or press Ctrl + I (or Command + I on Mac).
2. Message: Enter your query, for example: "**In the models.py file for the elements app, my Type model has errors. Please fix it.**"

And Gemini will detect the mistakes and correct them for you.

Introduction to migrations

With this, the next thing we are going to do is generate the tables of our database.

Initially, we are going to use SQLite, which is the one that is configured by default in a Django project.

From the project in Django, you are not going to connect directly to the database through SQL queries that we program; creating tables directly is not recommended when working with the framework and pretty much any modern server-side web framework follows this same principle of preventing

developers from interacting with the database directly, and this is great for two reasons:

1. It allows us to work with the database as one more process.
2. It allows us to perform better when working as a team.

Before explaining the two points in detail, let's mention what is the tool used to achieve such a feat... They are known as migrations, which are nothing more than a control system for the tables in the database.

In short, to start working with the models, we need the tables in the database, and to work with the tables, we need the migrations; let's start with migrations.

An interesting feature that Django has is that they do not use an additional file to define the migrations, otherwise we can use the same models as a template; therefore, we use the models both to define the structure of our entities, being the models layer, and also to define the tables in the database through migrations.

Migrations

Migrations are a system that allow us to generate tables based on classes that we create and the framework maps a class for each table in our database that we define; through migrations we can, or create new tables, or modify existing ones; a migration looks like the following:

```
migrations.CreateModel(
        name='Category',
        fields=[
          ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
          ('title', models.CharField(max_length=255)),
          ('slug', models.SlugField(max_length=255)),
        ],
),
```

This structure is automatically generated by the framework for us based on the models we have created, and it gives us the opportunity to modify the migration before reflecting it to the database; although, usually it is not necessary to modify it.

In the structure above, you can see each of the fields and their defined options such as maximum length, dates, primary key, and column types.

? Benefits of Using Migration

Basically, migrations allow us to work with the database as if it were just another process within the code.

- **SQL Abstraction**: We don't have to write or generate SQL code manually; we can focus directly on working with our project's logic.

- **Database Independence**: Our project is independent of the database we're using. In this case, it's SQLite, but we can easily configure PostgreSQL, MySQL, or any database supported by Django with just a few lines of code.

This mechanism, called migrations, is the system that allows us to generate these tables. When we run the command in the next lesson, you'll see that a file structure will be generated that will be used to apply these changes to the database.

Create migrations

In order to create the previous structure, based on our models, we have the following command:

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

Having two applications, we will have an output like the following:

```
Migrations for 'elements':
 elements/migrations/0001_initial.py
  - Create model Category
  - Create model Type
  - Create model Element
Migrations for 'comments':
 comments/migrations/0001_initial.py
  - Create model Comment
```

Django executes each of the migrations that we have defined in blocks and indicates what the model is; the order in which you run the migrations depends on the order in which we will install the apps in INSTALLED_APPS.

Or indicating the name of the applications for a time, and with this, which migrations we want to generate:

```
$ python manage.py makemigrations elements
```

Getting as output:

Migrations for 'elements':

```
elements/migrations/0001_initial.py
  - Create model Category
  - Create model Type
  - Create model Element
```

And

```
$ python manage.py makemigrations comments
```

Getting as output:

```
Migrations for 'comments':
  comments/migrations/0001_initial.py
    - Create model Comment
```

The 000X is the migration number; since, you can add or manipulate the existing model classes and with this create new migrations on these changes; they are generated incrementally. You can press Control + Click to open the corresponding file.

? Database Verification

We've created the structure. But, surprise! What if we check the database?

Notice that the database (db.sqlite3) is still empty, since we've only created the structure and haven't executed it yet.

Run migrations

At the moment, we have generated the migrations, that is, the structure that we showed previously:

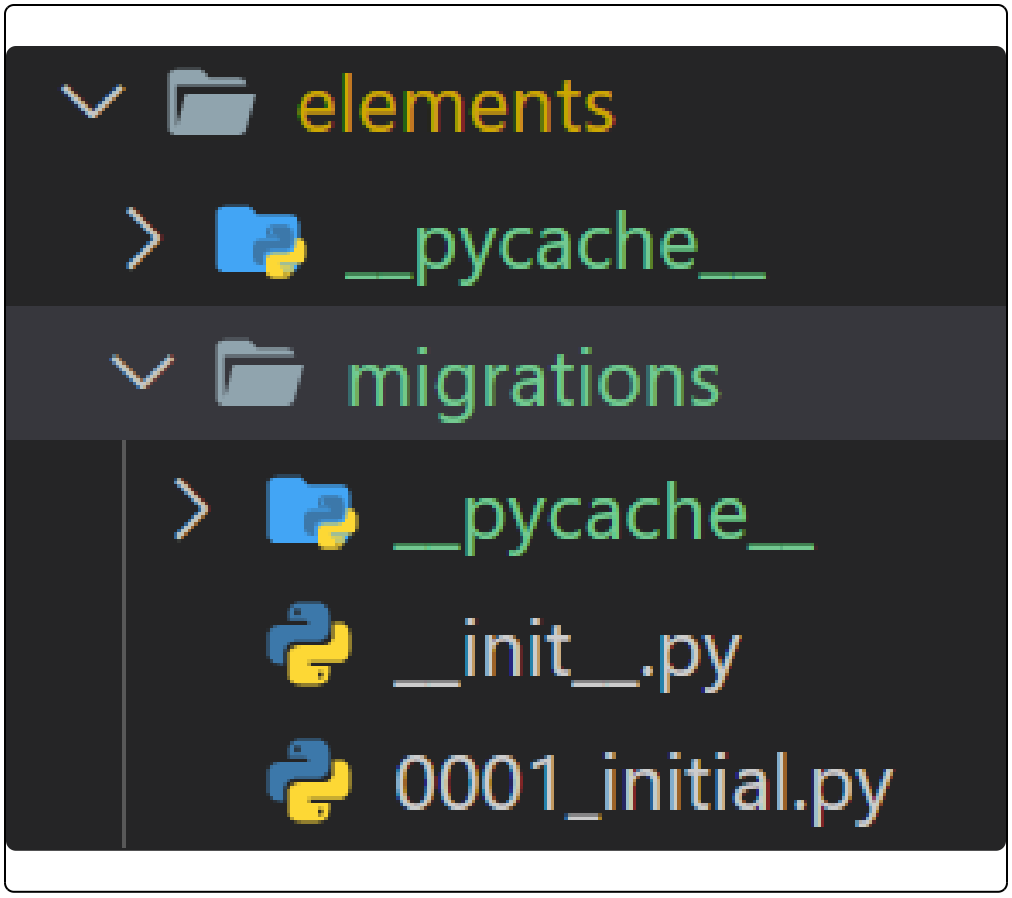


Figure 3-1: Initial migration for element application

And:

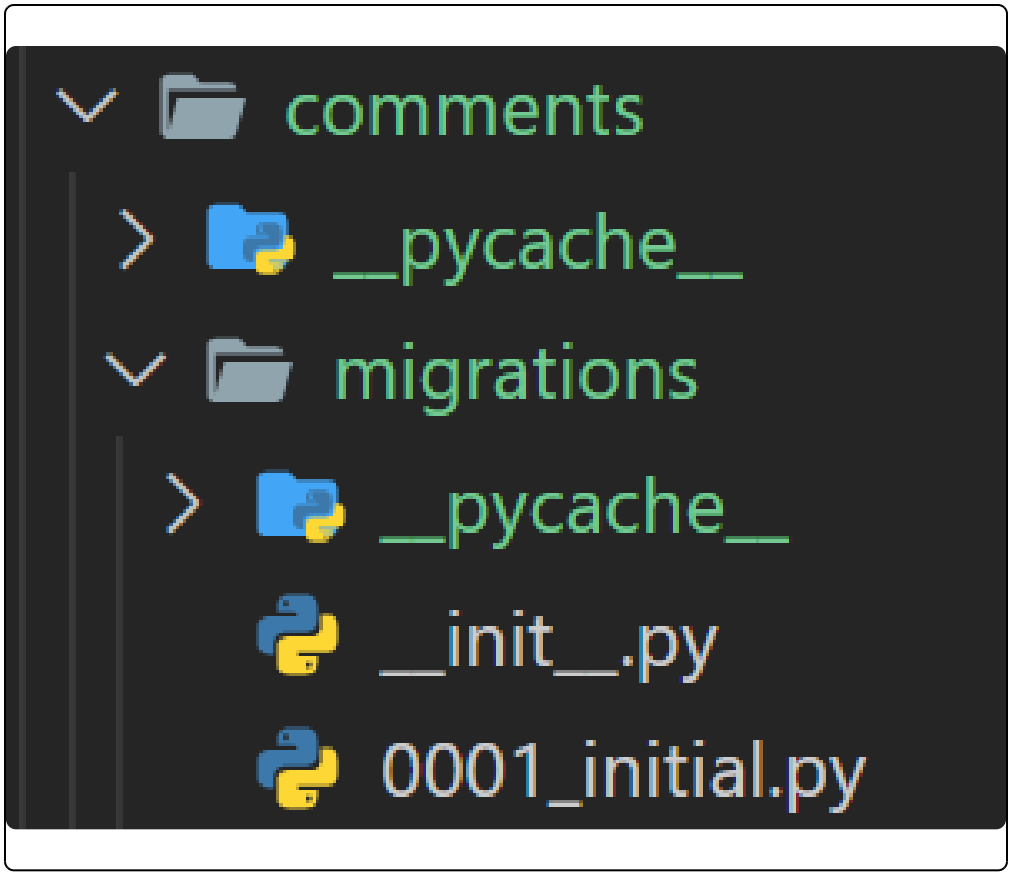


Figure 3-2: Initial migration for comment application

Now, we want to reflect it in the database; which can be any type of database supported by Django, such as PostgreSQL, MySQL or in this case SQLite:

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

And we will have an output like the following:

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying elements.0001_initial... OK
Applying comments.0001_initial... OK
Applying sessions.0001_initial... OK
```

It's important to understand this workflow:

- **makemigrations**: Creates the migration files (what we did earlier).

- **migrate**: Runs the migrations so they are reflected in the database.

If you haven't already done so, create and run your migrations as we did above.

Now we need to see the tables we generated earlier; To be able to see the structure of an SQLite database, I recommend the following program:

https://sqlitebrowser.org/

A simple manager to visualize the SQLite database; you can use any other, but in the book we will use the former.

SQLite databases are a single file; in the case of Django, this file is in the root of the project:



Figure 3-3: Database File in SQLite

and it is called as db.sqlite3.

And when opening our database with SQLite Browser (From the button called "Open Database" located in the header):

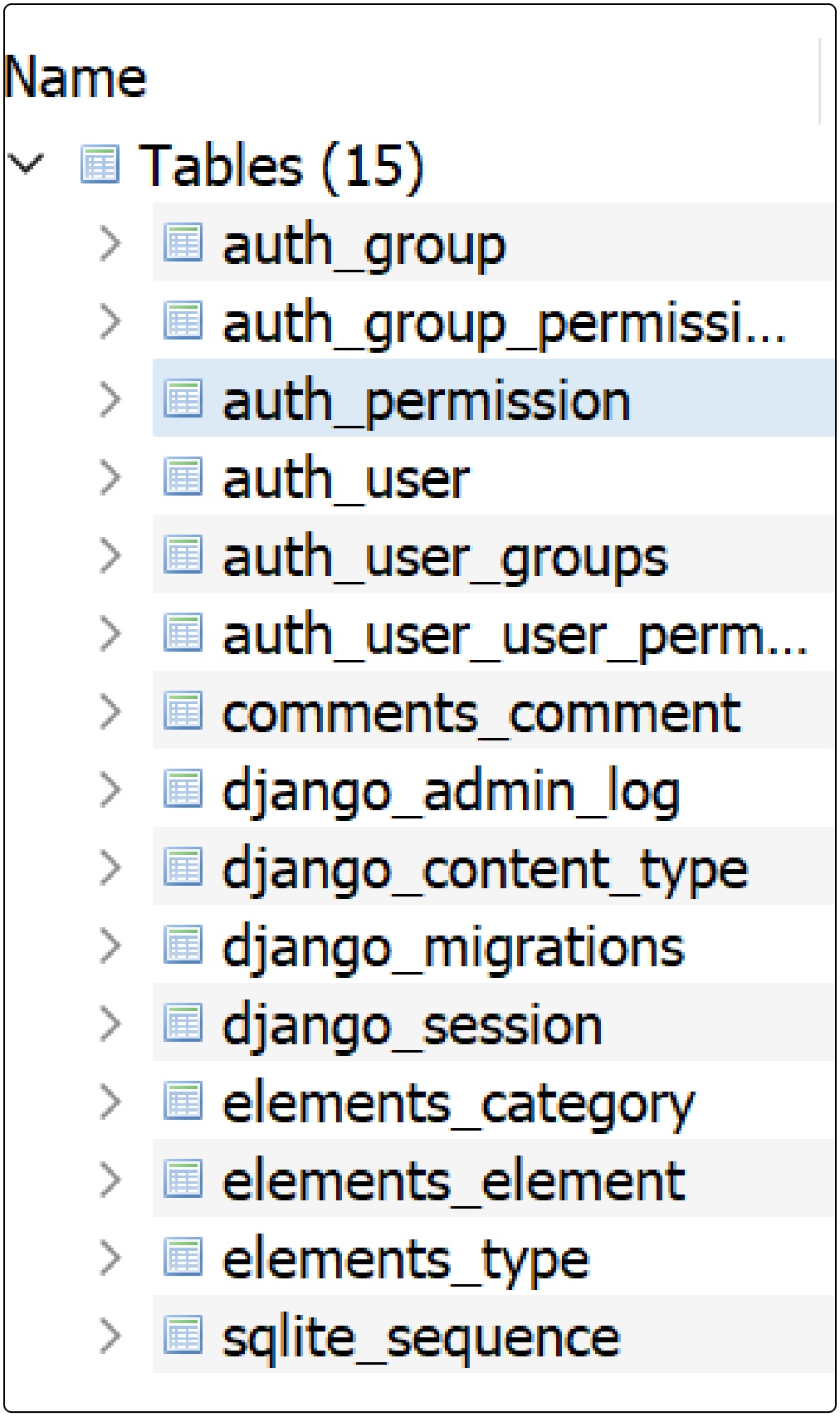

Name
Tables (15)
auth_group
auth_group_permissi...
auth_permission
auth_user
auth_user_groups
auth_user_user_perm...
comments_comment
django_admin_log
django_content_type
django_migrations
django_session
elements_category
elements_element
elements_type
sqlite_sequence

Figure 3-4: Created tables

Note that we have a lot of tables with compound names; in these names we can see a prefix that is repeated:

- auth_*
- django_*
- elements_*
- comments_*

This is an organization that uses the framework internally to know which tables belong to which application, and we can clearly know which table belongs to us; this is an organization internal to the framework, and we don't have to worry about this prefix.

These prefixes indicate which application it belongs to; Django is made up of our own applications, developed by us, as well as others that come by default with the framework; for example, the user's, which allows to handle authentication (django.contrib.auth)

It's important to note that we have many Django-internal tables that of course have a particular purpose:

- auth_group
- auth_group_permissions
- auth_permission
- auth_user
- auth_user_groups
- auth_user_user_permissions

- django_admin_log
- django_content_type
- django_migrations
- django_session

Which are for the exclusive use of the framework and allow registering users, permissions, sessions, migrations, among others

?? The django_migrations Control Table

There is an important table that is generated: **django_migrations**.

- **Purpose**: This is a control table that indicates which migrations have been executed and which have not.
- **Test**: If you run **python manage.py migrate** again, it will tell you that there is nothing to migrate. This is because it links the migrations in your project to the records in this table and determines that everything is already up to date.

Revert migrations

We can also reverse migrations; this is particularly useful if we want to change the model and run everything again; for that:

```
$ python manage.py migrate <applicationName> <migrationNumber>
```

For example:

```
$ python manage.py migrate comments 0001
```

What I would do is remove any migrations above 0001 for the comments application; since we don't have any, let's make some subtle changes to:

comments/models.py:

```
from django.db import models
from elements.models import Element
# Create your models here.
class Comment(models.Model):
    text = models.TextField()
    date_posted = models.DateTimeField(auto_now_add=True)
    element = models.ForeignKey(Element, related_name='comments',
on_delete=models.CASCADE, null=True)

    def __str__(self):
        return 'Comentario #{}'.format(self.id)
class Test(models.Model):
    text = models.TextField()
```

We execute:

```
$ python3 manage.py makemigrations comments
```

We have as output, the generation of the new migration:

```
Migrations for 'comments':
  comments/migrations/0002_test.py
    - Create model Test
```

And if we reverse:

```
$ python manage.py migrate comments 0001
```

We have as output that migration two was reverted, and any that could be defined above it:

```
Operations to perform:
  Target specific migration: 0001_initial, from comments
Running migrations:
  Applying elements.0001_initial... OK
  Applying comments.0001_initial... OK
```

In short, migration 0002, which corresponded to the so-called “test”, was eliminated from the database.

To finish, delete the test class and the migration of 0002_test.py that we will not need.

...

Chapter 5: Django Admin

One of the most powerful parts of Django is the interface to automatically manage records with little configuration.

Django's admin panel is one of its most powerful features and a prime example of its "batteries included" philosophy. It's a self-generating module that provides a complete interface for managing your application's data without writing any HTML, CSS, or JavaScript code.

Django Admin is nothing more than an application configured by default in a Django project that integrates perfectly.

Django Admin, reads metadata from models to provide an interface for managing site content; that is, the typical CRUDs.

?Create superuser

To be able to enter it, we have to create a superuser; for it:

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

Which correspond to steps such as the following:

```
Username (leave blank to use 'andrescruz'): admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Once created, we start the server:

```
$ python manage.py runserver
```

Advantages of Superuser Access

- **Full Access**: You have complete control over all parts of the system.
- **Basic Authentication**: Django already includes basic authentication features (roles, permissions, and user management), although for a registration and login system for regular users (non-administrators), you would need additional modules or implement it yourself.

To access the administration panel (located at the URL /admin), we need a Superuser.

http://127.0.0.1:8000/admin/

Django Admin in practice

We log in with the user we created earlier and when entering we will see:

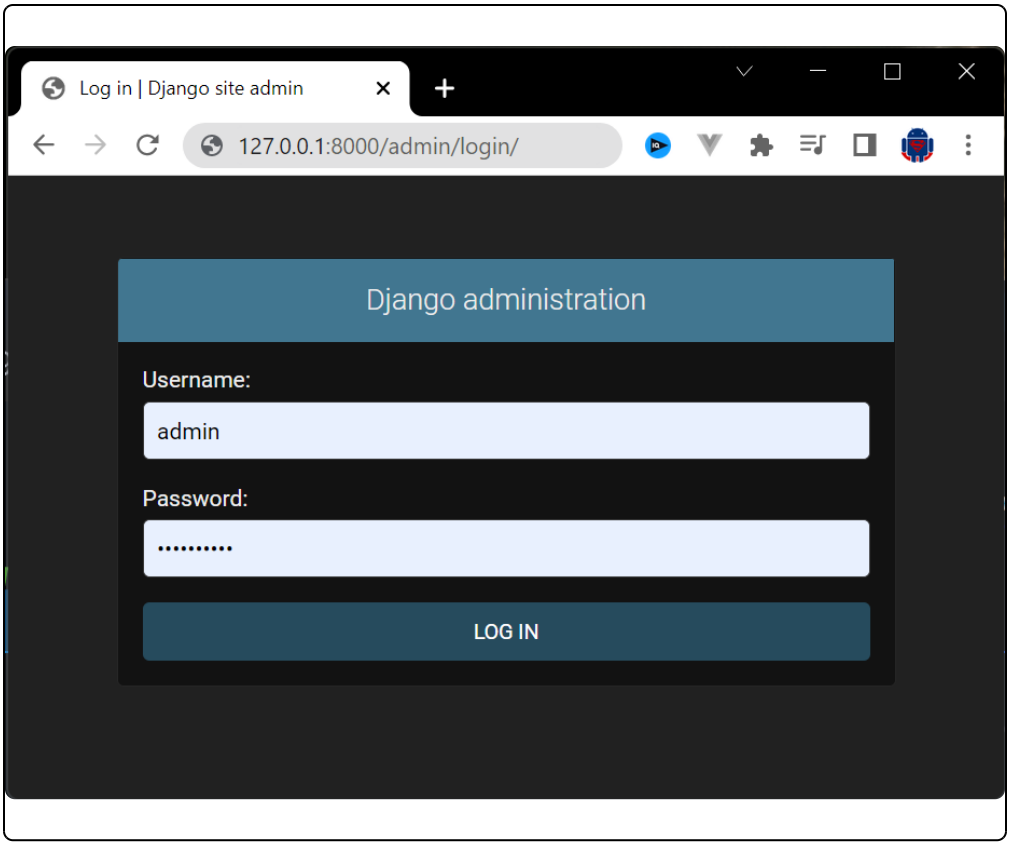


Figure 4-1: Login page

Once we've created the Superuser, we can access the administration panel. Although its design is functional and outdated, it offers robust, ready-to-use management of users, groups, and permissions.

In this section, we have multiple administration sections, by default, we have access to manage:

- Groups
- Users

Enter each of these options:

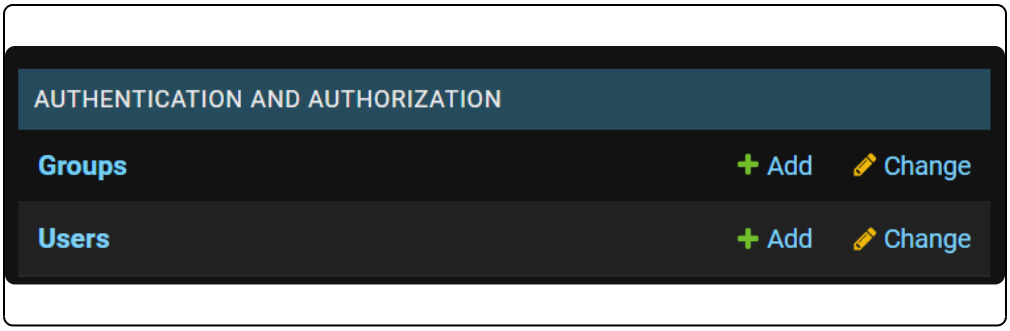


Figure 4-2: CRUD of users and groups

If you click on "Add" in the "Groups" section:

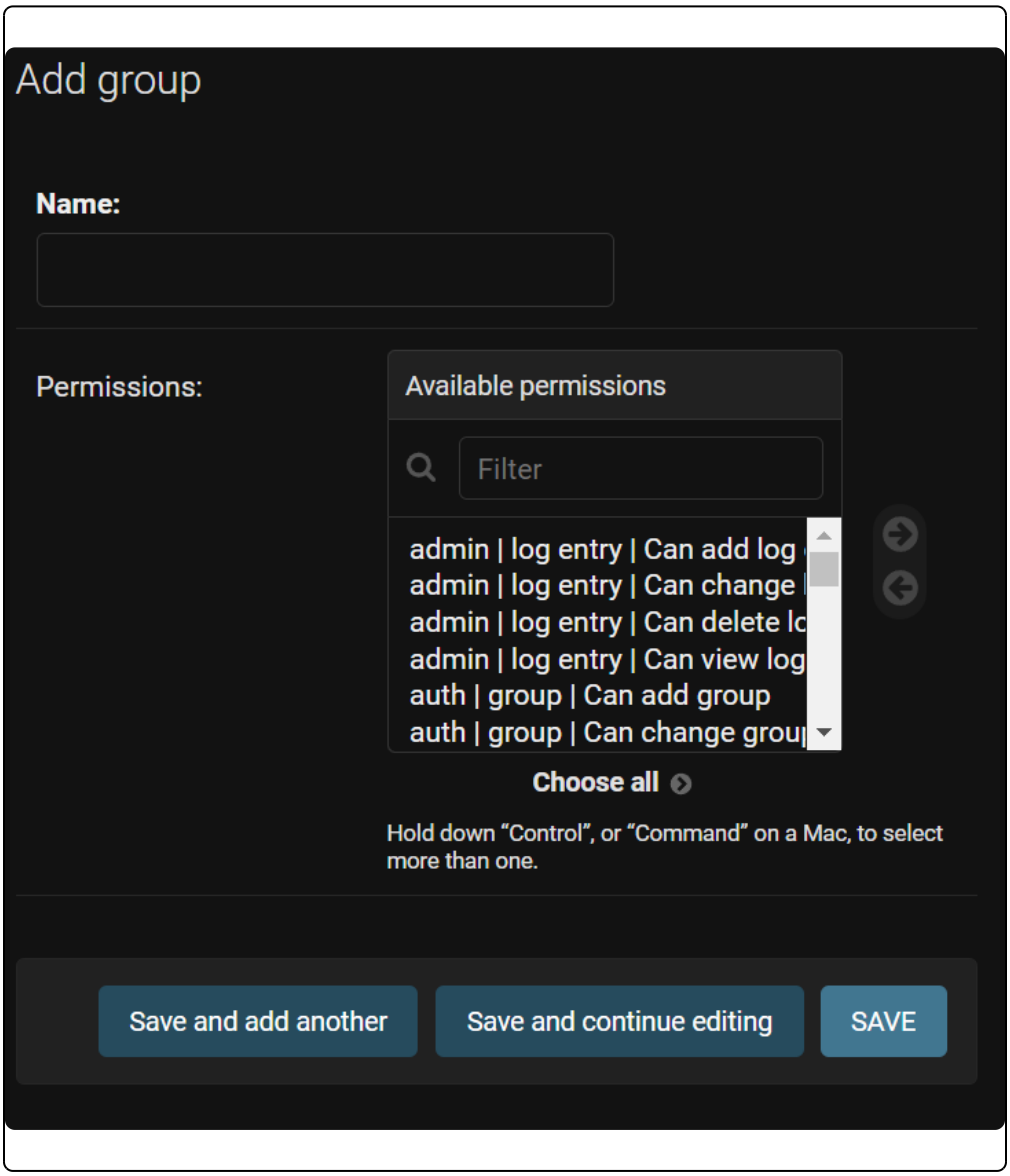


Figure 4-3: Add a group

Permission Structure: For each model you register (e.g., Comment in your application), Django automatically creates four basic permissions:

- **Can add**: Permission to create records.
- **Can change**: Permission to update records.
- **Can delete**: Permission to delete records.
- **Can view**: Permission to view the list and details.

Group Structure: functions as a centralized Roles system to facilitate permission management.

- **Functionality**: Instead of manually assigning the same 10 permissions to 50 users, you can create a Group (e.g., "Comment Managers"), assign the 10 necessary permissions to that group, and then simply assign the users to that group.
- **Advantage**: It reduces administrative complexity, since any user added to the group will automatically inherit the associated permissions.

In short, we have a management module for groups; the reason that these options appear to manage, is that these entities are part of a project in Django.

Add our own relations

From here, we have the typical CRUD available that we can create manually as we did before, but with the advantage of being automated.

However, in the image above, we don't see the comments entity; since, to use it, we have to register it in the admin.py file at the level of application; in this case, the comments model:

comments/admin.py

```
from django.contrib import admin
from .models import Comment
# Register your models here.
admin.site.register(Comment)
```

Explanation of the above code

In order to manage our own models, we use the register() method and indicate the model; already with this:

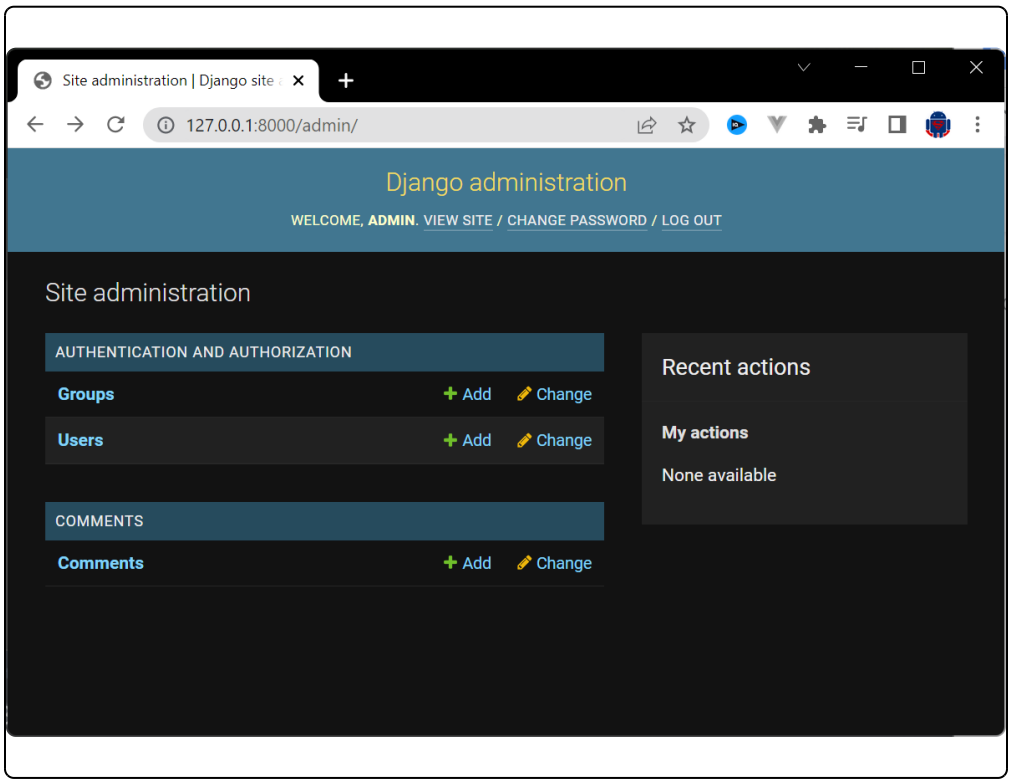


Figure 4-4: Register a custom model

From the "Add" option in any of the assigned models, we have the possibility to add a comment:

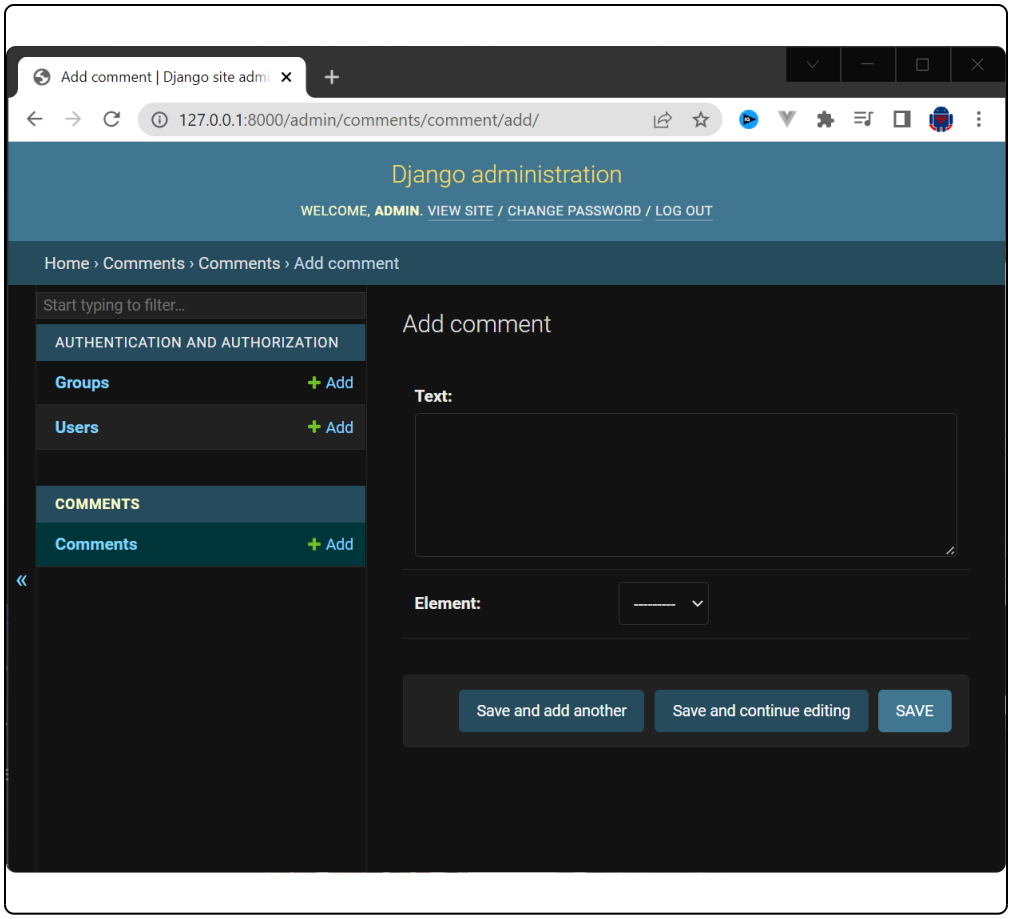


Figure 4-5: Add a comment

Since " Change " allows us to both update records:

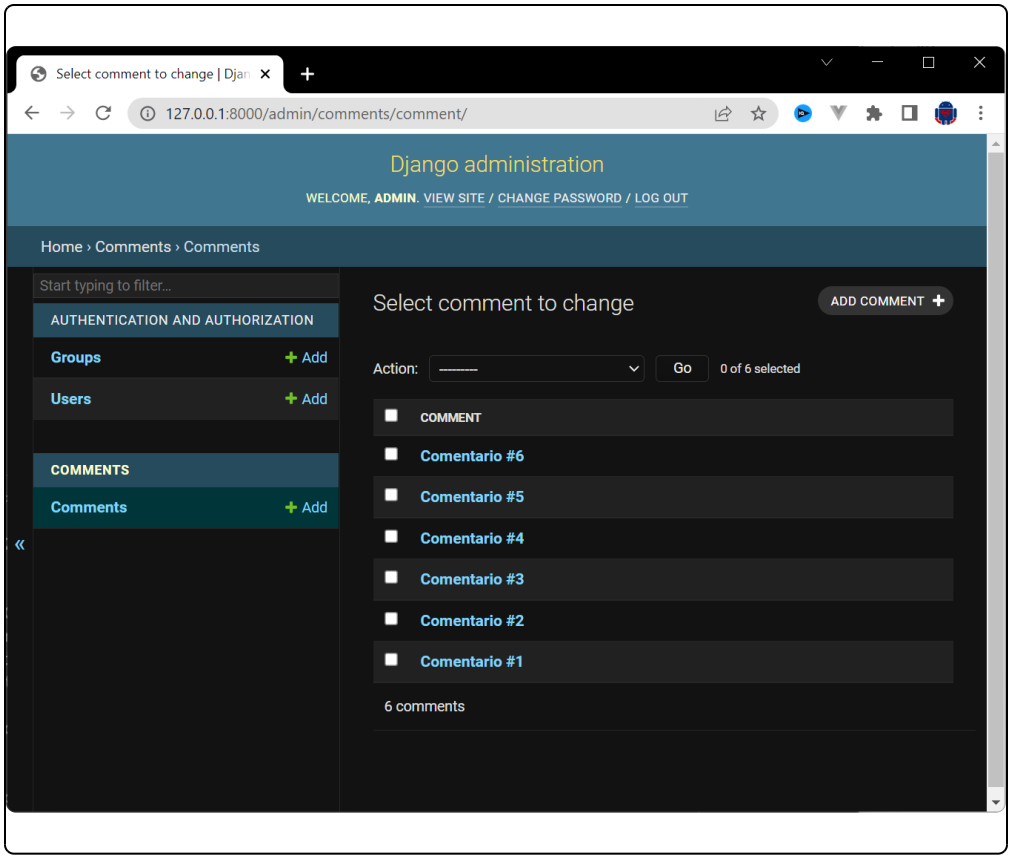


Figure 4-6: List of comments

How to remove them:

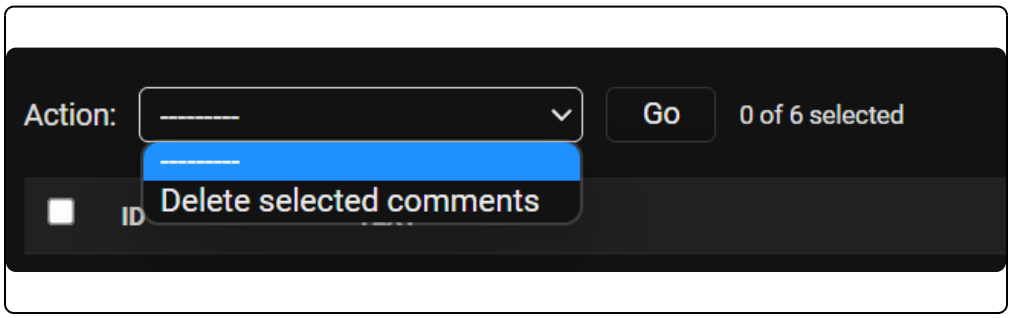


Figure 4-7: Delete comments

This representation of the comment as "Comment #X" is due to the fact that in the model we have:

comments/models.py

...

Chapter 6: Forms

Previously, we saw how to use the form of type model or also called ModelForm; but there are others that are more manipulable and extensible than these, but they are also more complex to use and require more code.

Regardless of the type of form we use, forms are a very present structure in web development and the scheme used par excellence to interact with the user, and change the data models through them. In between chapters, we will learn more about how to use forms and we will learn about the two types available in Django, ModelForm and forms.

? Two Types of Forms in Django: ModelForm vs. Form

In Django, there are primarily two types of forms for handling user interaction and data management. We are already familiar with one, but it is crucial to know the other for more complex use cases.

1. ?? ModelForm (Model-Bound Forms)

ModelForms are forms that are directly tied to a database model.

Advantages:

- **Automatic**: They are perfect for creating simple CRUD (Create, Read, Update, Delete) processes, such as those we have used for comments, posts, or categories.
- **Direct Relation**: The field definition is automatically extracted from the model, which reduces code (you simply declare the fields tuple or __**all**__).
- **Implicit Validations**: They automatically inherit the validations defined in the model.

Disadvantages:

- **Limitation**: They are very tied to a single entity.

2. ? Form (Standard Forms)

Forms (normal Django forms) are forms that are not associated with any specific database model.

Advantages:

- **Flexibility**: They are more generic and give you 100% control over every field you wish to display.
- **Complex Cases**: Ideal for complex scenarios, such as filling two or three different entities on a single screen, or displaying fields that are not defined in any particular model.

Disadvantages:

- Greater Effort: They are more manual, which translates into more code, more work, and a higher propensity for errors.

Django ModelForm

If you are creating an application that has a very close relationship with the database, and this with its models, this type of forms is particularly useful, which, as its name indicates, has a direct relationship between the models and the forms.

ModelForms allow you to define a close relationship between models and forms; specifically with the ModelForm we avoid redefining the fields at the form level that we already have defined in the model; having as an extra advantage, that they inherit the data types, lengths, if they are required fields, etc.

Official documentation:

https://docs.djangoproject.com/en/dev/topics/forms/modelforms/

Django Forms

These allow us a higher level of customization, but sacrificing the automation of tasks, since we have to create validation rules, specify the fields and types of fields, the saving process... and in short, they are the option if the form that we want to implement is not possible (or complicated) with the **ModelForm** type.

Forms in Django have different field types for different types in HTML:

- Numeric types.
- Texts.
- Regex for regular expressions.
- Selection.

Among others you can review:

https://docs.djangoproject.com/en/dev/ref/forms/fields/

Each of these fields contain options to customize them, either to add validations, the label, required, minimum and maximum lengths, **QuerySet** initial value among others.

Finally, a form field using Django's forms looks like the following:

```
title = forms.CharField(label="Title", max_length=255, min_length=3)
```

Here is the official documentation:

https://docs.djangoproject.com/en/dev/topics/forms/

Which one to use depends on the development requirements you want to do; a rule could be to use the ModelForm whenever possible, since it is the one that offers us the most direct approach (but also the most closed or difficult to customize), if you see that the model you want to manage has very particular requirements, then Django forms will be a better option.

Case study

Explained what Django forms are for and how they differ from Django **FormModels**, we're going to want to manage elements, in the elements application; so, we create a file and use the following definition:

Form

elements/forms.py

```
from django import forms
from .models import Type, Category
class ElementForm(forms.Form):
   title = forms.CharField(label='Title', max_length=255, min_length=3)
   slug = forms.SlugField(label='slug', max_length=255, min_length=3)
   description = forms.CharField(label='Description',initial='Content initial',
widget=forms.Textarea)
   price = forms.DecimalField(label='Price', decimal_places=2, max_digits=5, required=
False)
   type= forms.ModelChoiceField(label='Type', queryset=Type.objects.all(), initial=1)
   category= forms.ModelChoiceField(label='Category', queryset=Category.objects.all())
```

Explanation of the above code

We are going to explain one by one the fields that we created previously:

1. The title field we use a **CharField** which is a text field and we indicate as the options:
2. label for the **HTML LABEL** tag.
3. **max_length** maximum allowed length.
4. **min_length** minimum allowed length.
5. The description field we use a **CharField** and indicate the options:
6. Initially we indicate a default value (we set this option purely for demonstration purposes).
7. widget with this option we indicate additional options such as the type of field, classes or attributes that we want the form field to have.

8. The rest of the options presented above
9. In the price field we indicate the price of the element (remember that each element can be some type of ad that we are selling) we define a **DecimalField** type field and it has the following options:
10. **required** to indicate whether or not the field is required.
11. **max_digits** the maximum number of digits.
12. **decimal_places** we indicate the maximum number of decimals.
13. The type field, we indicate the types and for that we use a field of type **ModelChoiceField** with the options of:
14. With the **queryset** we can make the query to the database that will bring the records to create the selection fields for the SELECT.
15. The category field follows the same guideline as the type field, but by querying the categories using a queryset.

You can see the full list at:

https://docs.djangoproject.com/en/dev/ref/models/fields/

Key points:

- As with everything in Python and Django, the form is defined as a class. In this case, it inherits directly from **forms.Form**, not **forms.ModelForm**.
- For fields that are relationships in the model (such as **ForeignKey** or **ManyToManyField**), the **ModelChoiceField** type is used in the standard form. This field is rendered as a select element.

- The **queryset** defines the database query used to populate the field's options.
- Ultimately, the two form types have many similarities, but their implementation differs in the aspects that define each. For example, **Form** defines the fields (whose classes, like **CharField**, have the same names as in the models), but **ModelForm**, due to its direct relationship with the models, does NOT define these fields.

View for the form

Let's create a function to create an element:

elements/views.py

```
from django.shortcuts import render
from .forms import ElementForm
from .models import Element
def add(request):
    if request.method == 'POST':
        form = ElementForm(request.POST)
        if form.is_valid():
            element = Element()
            element.title = form.cleaned_data['title']
            element.slug = form.cleaned_data['slug']
            element.description = form.cleaned_data['description']
            # element.price = form.cleaned_data['price']
            element.category = form.cleaned_data['category']
            element.type = form.cleaned_data['type']
            element.save()
    else:
        form = ElementForm()

    return render(request, 'element/add.html', {'form':form})
```

Explanation of the above code:

In the **add()** view function:

- GET type request: We create the form and pass it to the template:

```
form = ElementForm()
return render(request,'element/add.html',{'form':form})
```

- **POST** type request: We process the form and create a record (element) from the data passed in it; it is important to note that, on this occasion, we cannot save the record directly from the form, otherwise we have to access each of the fields independently and set them in an object of the model class.
- Regarding the price, it is commented that if it is not specified in the form by the client, then it gives an error that the price cannot be null, therefore, the price takes the default value of 6.10.

With cleaned_data we get the data supplied in a form field; this property can only be used after executing:

```
form.is_valid()
```

This function validates the form against specific validations and creates safe/clean access to the data we query through cleaned_data; if you try to access this property without first calling the is_valid() function, it will give an error like the following:

```
'ElementForm' object has no attribute 'cleaned_data'
```

We create the URL:

elements\urls.py

```
from django.urls import path
from .views import add, index, update, delete
app_name = 'elements'
urlpatterns = [
    path('', index, name='index'),
    path("add", add, name="add"),
    path('update/<int:pk>', update, name='update'),
    path('delete/<int:pk>', delete, name='delete'),
]
```

And we register at the Django project level:

mystore\urls.py

```
urlpatterns = [
    ***
    path("elements/", include("elements.urls")),
]
```

Since we're using a standard form (**Form**) and not a **ModelForm**, the advantage of automation is lost.

We now have to manually create the model instance and assign each field:

- We get the form data (**form.cleaned_data**).

- We create the model instance (**element = Element()**).

We assign the values one by one (e.g., **element.title = form.cleaned_data['title']**).

Template for the form

Let's create the template to create the previous view function:

elements/templates/element/add.html

```
<form action="{% url 'elements:add' %}" method="post">
  {% csrf_token %}
  {{ form }}
  <input type="submit" value="Enviar">
</form>
```

...

Chapter 14: Django Channels

Nowadays, there are multiple applications that require real-time communication functionality; for messaging applications such as chats, this type of functionality is essential to keep clients connected so they can send and receive messages.

Of course, you can implement this real-time functionality on a standard server using technologies like AJAX, but obviously, constantly sending requests to our server to check the status is extremely inefficient; for these cases, there is a mechanism based on WebSockets that allows us to implement this real-time feature efficiently. In Django, for this purpose, we have what is known as Django Channels, which allows us to implement WebSockets in our application through a channel layer.

The link to the official documentation is as follows:

https://channels.readthedocs.io/en/latest/

?? Django Channels and WebSockets

In order to use WebSockets and their full-duplex communication, we have to change our default WSGI-based server to an ASGI-type server, which is what Django Channels provides.

The Problem with Traditional "Real-Time"

Imagine a chat like WhatsApp or Telegram. If you send a message, the server receives it (Request → Response). But how does the other person find out they have a new message? Under the traditional

scheme, you would have to use an inefficient JavaScript "trick" called Polling (with setTimeout or setInterval):

- Client: "Are there new messages?" -> Server: "No".
- Client: "Are there new messages?" -> Server: "No".
- Client: "Are there new messages?" -> Server: "Yes, here is one".

This approach is a waste of resources: the client consumes battery and data, and the server processes thousands of useless requests.

? The Solution: Django Channels and WebSockets

Django Channels allows the server and the client to maintain a permanently open "channel." Once the connection is established, the server can send data to the client at the exact moment it occurs (for example, when a message arrives from another user), without waiting for the client to ask.

WSGI and ASGI Servers

In this section, we are going to talk about the types of servers available in Django: WSGI servers, which are synchronous servers like the ones we have been using so far, or ASGI servers, which are asynchronous and are used when we want to implement full-duplex communication between the client and the server.

Synchronous Applications using WSGI (Web Server Gateway Interface)

The default server we use when deploying or publishing our application is generally a WSGI type, which is the definition web servers take when developing in Django; in other words, WSGI is to Python what Apache is to PHP.

WSGI stands for Web Server Gateway Interface, which are synchronous servers; therefore, we CANNOT implement real-time features or full-duplex communication.

What is Full Duplex and why do we need it?

Normally, the web works under the HTTP protocol, which is unidirectional. This means the client (your browser) sends a request to the server, Django processes it, and returns a response. The cycle ends there. The server cannot talk to the client if the client did not ask something first.

Asynchronous Applications using ASGI (Asynchronous Server Gateway Interface)

WSGI servers in Python, much like Apache, are used to handle HTTP requests and work synchronously with our request; they cannot work asynchronously, and for these cases, we need another type of server called ASGI.

As mentioned earlier, WSGI servers are the standard for developing in Python; however, to develop asynchronous applications in Django, we need an ASGI server that allows us to handle the traditional HTTP request scheme as well as WebSockets through a layer of channels.

Django Channels allows us to use an ASGI-type server, enabling us to handle not only HTTP requests but also protocols that require end-to-end communication, that is, persistent full-duplex communication by opening a bidirectional channel to transmit messages using the TCP (Transmission Control Protocol). This full-duplex communication allows our client and server to connect persistently without the need to check status via AJAX.

WSGI: Classic/Synchronous Connection

When an HTTP request is sent by a client through a browser to our web server, Django handles the request by passing it to an HttpRequest object and sends it to the view according to the configured route; the view processes the request and returns an **HttpResponse** object, sending it back to our client (the web browser) according to the following scheme:

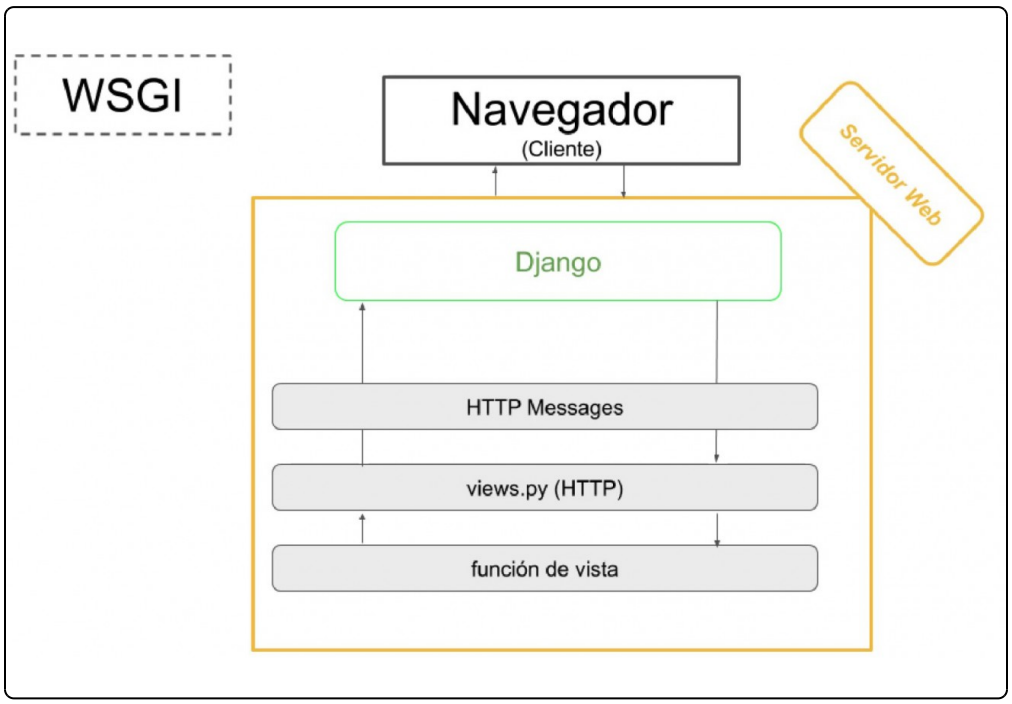


Figure 17-1: WSGI Server Process

ASGI: Asynchronous/Synchronous and Full-Duplex Connection

Channels in Django replace the previously mentioned scheme through messages passed end-to-end asynchronously in our application; HTTP requests are still mapped through the same routing scheme we have for a Django project, but to handle channel routes (i.e., WebSockets), we need another layer, a different set of routes to handle these full-duplex requests. Therefore, we can use a synchronous architecture, an asynchronous one, or even both using ASGI servers via Django Channels:

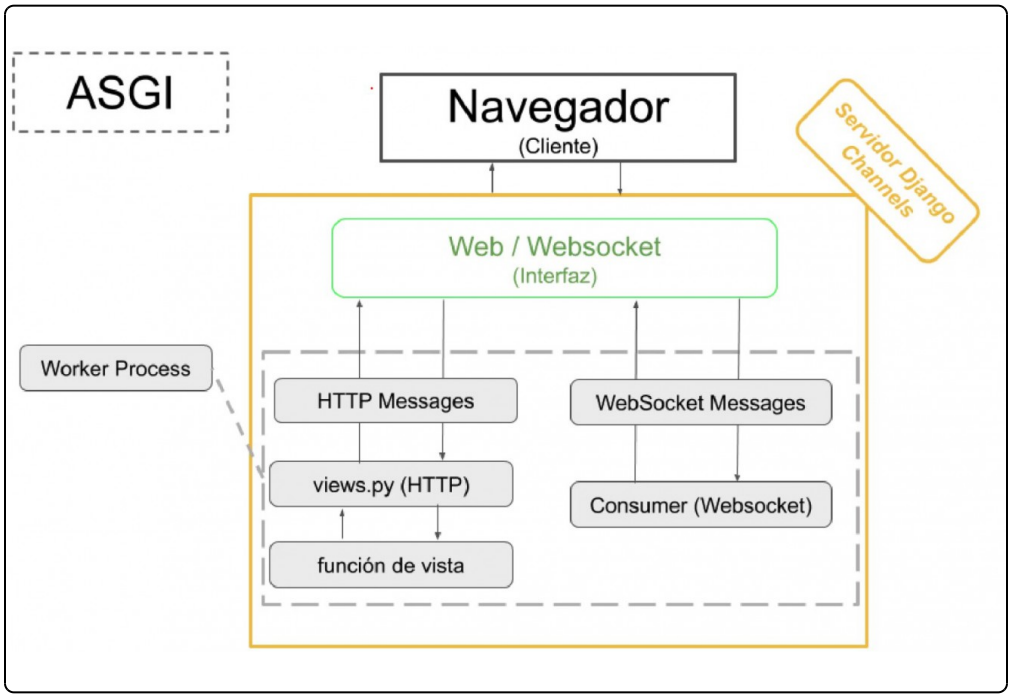


Figure 17-2: ASGI Server Process

Since changes in server state will only occur when messages are sent and/or received, it would not be necessary to use the AJAX scheme required in synchronous WSGI servers.

How does the Channels architecture work?

Summary

In an application with Django Channels, two worlds coexist:

- **Synchronous World (HTTP)**: For loading the page, viewing a profile, etc. It is handled with the views (**views.py**) you already know.
- **Asynchronous World (Channels)**: For chat or real-time notifications. Here we don't use "views," but **Consumers**. A Consumer is like a chat room where users connect and stay there listening for messages; we will discuss this in detail later.

Summary

At the beginning of the course, we saw two files in the Django structure that now make sense:

- **WSGI (Web Server Gateway Interface)**: The traditional Python standard for synchronous web applications (HTTP). It is what we have used until now.
- **ASGI (Asynchronous Server Gateway Interface)**: The spiritual successor to WSGI. The "A" stands for Asynchronous. This standard allows handling not only HTTP but also WebSockets, which enable bidirectional and real-time communication.

Set up project in Django with Django Channels

In this section, we are going to create a minimal application with Django Channels to send messages between clients who will be registered in a room where they can communicate, therefore it will be a group chat as if it were a group on WhatsApp, Telegram or Discord; to do this, we will create a new project as discussed in chapter 3 that we will call mychannels.

Basic installations and configurations

Let's start by preparing our working folder for this new project focused on Django Channels.

- **Creating the Virtual Environment**: We create a new folder and configure a clean virtual environment to avoid version conflicts.
    - We use **python -m venv .venv** (or your preferred tool).
    - We activate the environment and verify that it is empty using **pip freeze**.
- **Installing Django**: We install the latest version of Django:
    - ```
      $ pip install django
      ```

We install Django channels using:

```
$ python -m pip install -U 'channels[daphne]'
```

Configure ASGI server and Django Channels
```

As we mentioned previously, we will use Django Channels along with an ASGI type server that we must configure; in the official documentation, they specify the steps that we exemplify in this section:

https://channels.readthedocs.io/en/latest/installation.html

1. Configuring asgi.py

To enable Django to handle **Consumers** (which we'll see later, but basically they allow asynchronous requests), we need to configure protocol routing.

Originally, our **asgi.py** file only handled HTTP. Now we're going to expand it to also recognize WebSockets.

We prepare the project to use the ASGI type server and keep the HTTP channel:

mychannels/asgi.py

```
import os
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application
from django.urls import path
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()
application = ProtocolTypeRouter({
  # Django's ASGI application to handle traditional HTTP requests
  "http": django_asgi_app,
  # WebSocket chat handler
  "websocket": AllowedHostsOriginValidator(
    AuthMiddlewareStack(
      [
        # URLRouter(chat.routing.websocket_urlpatterns)
      ]
    )
  ),
})
```

With:

```
"http": django_asgi_app,
```

We preserve the HTTP channel, therefore, we can use the same project in Django to communicate applications through websockets using the routes defined in:

chat\routing.py

And the traditional routes defined in the Django project of synchronous type that we define in urls.py at the level of the Django applications.

It's important to mention that the routing files for consumers/wetsockets haven't been configured yet, so you could comment out the following line:

2. Point it to settings.py

It is important to mention that the route files for consumers/wetsockets have not yet been configured, so you could comment the following line:

...

Chapter 16: Introduction to Testing

Testing is a crucial part of any application we are going to create, regardless of the technology, it is always advisable to perform automatic tests to test the system when new changes are implemented; in this way we save a lot of time since there is no need to perform many of the tests manually but by executing a simple command.

Testing consists of testing components individually; in the case of the application we have built, they would be each of the API methods, as well as any other dependency on these methods; this way, when these automated tests are run, if the application passes all the tests, it means that no errors were found, but, if it does not pass the tests, it means that changes have to be made at the application level or Tests implemented.

Why test?

Testing helps ensure that your application will function as expected and as the application grows in modules and complexity, new tests can be implemented and current tests adapted.

It is important to mention that the tests are not perfect, that is, even if the application passes all the tests, it does not mean that the application is error-free, but it is a good initial indicator of the quality of the software. Additionally, testable code is generally a sign of good software architecture.

Testing must be part of the application development cycle to guarantee its proper functioning by running them constantly.

Why Learn Testing in the Age of AI?

Today, with the rise of Artificial Intelligence, implementing tests is easier than ever. AI can handle 80% to 90% of the work involved in writing test code. However, to use these tools correctly, you first need to understand the basics.

I recommend you don't skip this section; we'll cover the essential operations so you understand how to test your components and why it's a time investment that pays for itself.

What are tests for?

Testing involves evaluating individual components to ensure that, when combined, the entire system functions correctly. With tests, we can:

- **Simulate HTTP requests**: Test form submissions (POST) or page loads (GET).
- **Validate the database**: Verify that records are created, edited, or deleted as expected.
- **Evaluate behavior**: Confirm that redirects work, that security validations prevent unauthenticated users, and that errors are handled gracefully.
- **Save time in the long run**: When adding new features, simply running a command will tell you if you've broken something that was already working (regression testing).

One of Django's greatest advantages is its modularity; everything is a separate component—one for views, another for forms, models, and so on. In testing, we can evaluate the following in isolation:

- **View responses**: Status codes (200 OK, 404 Not Found, etc.).
- **Business logic**: Signals, model methods, and utilities.
- **Forms**: Data validation and field cleaning.

What should we try?

Testing should focus on testing small units of code in isolation or individually.

For example, in a Django application or a web application in general:

- Views
- View responses
- State codes
- Nominal conditions (GET, POST, etc.) for a view function
- Forms
- Individual help functions

You can get more information in the official documentation:

https://docs.djangoproject.com/en/dev/topics/testing/

Limitations of Testing

It's vital to understand that testing isn't infallible. Even if all your tests pass, this isn't an absolute guarantee that the project is 100% error-free. What it means is that the scenarios you designed work well, but there may always be edge cases that weren't covered.

Understanding the first Tests

So that the use of Tests is understood in a less abstract way, we are going to create a small exercise of mathematical operations before starting to create Tests to test modules of our application, such as views.

In Django, we have very easy use of tests, if you notice, already in the applications we have a file called tests.py, which is to create our tests:

- comments.tests.py
- elements.tests.py

It is also not necessary to install any separate library, since as we indicated before, Django in its "Batteries Included" philosophy, brings everything necessary to create complete applications without the need to install additional dependencies, at least for the basic elements such as the tests that should be part of any development.

To know the basic operation of tests, we are going to create a test, in Django, at the application level, we have a file called tests.py in which we place the tests for the application; for this exercise, you can use any project in Django; in the tests.py file of a Django application, we create some functions that solve the mathematical operations:

<ProyectDjango><AppDjango>\tests.py

```
def add(a: int , b: int) -> int:
    return a + b
def subtract(a: int, b: int) -> int:
    return b - a
def multiply(a: int, b: int) -> int:
    return a * b
def divide(a: int, b: int) -> int:
    return b / a
```

At the moment, they are not tests intended for application, but rather basic mathematical operations, but with them we will be able to better understand how tests work; now, in the previous file, we define the following code, which correspond to the tests:

<ProyectDjango><AppDjango>\tests.py

```
***
assert add(2,2) == 4
assert subtract(5,2) == 3
assert multiply(10,10) == 100
assert divide(100,25) == 4
```

As you can see, there are only 4 lines of code in Python, but, using the assert keyword that is used to verify the outputs; specifically, we will be testing that the arithmetic operations are equal to the expected values, that is, if for the addition operation, we add `1+1`, the expected result is 2.

Finally, to test the previous tests we run:

```
$ python manage.py test
```

To run all the tests, which in this case corresponds to only one file; we will have an output like the following:

```
Found 0 test(s).
System check identified no issues (0 silenced).
----------------------------------------------------------------------
Ran 0 tests in 0.000s
OK
```

In which, as you can see, NOTHING was executed:

```
Found 0 test(s).
```

For Django to recognize tests, they must be inside a class that inherits from **TestCase** and implemented in methods that begin with **test_**:

```
class OperationTest(TestCase):
    def test_op(self):
        assert add(3,3) == 6
        assert add(5,4) == 9
        assert subtract(5,2) == 3
        assert multiply(10,10) == 100
        assert divide(100,1) == 100
```

Now if you run:

```
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

As you can see, all operations were executed satisfactorily, therefore, the application passed the tests;

One factor that you must take into account is that, **although the application passes the tests, it does not guarantee that it does not have errors, simply that not enough tests were done or defined to test all possible scenarios**; even so, it is a good meter to know the status of the application and repair possible problems.

If, for example, we place an invalid output when doing the tests:

```
def test_add() -> None:
    assert add(1, 2) == 4
```

...