









Complete Beginner's Guide to Django

Complete Beginner's Guide to Building Your First Django Project








-  Table of Contents
-  What We're Building
-  Understanding the Tools
 - What is a Terminal?
 - What is Git?
 - What is a Code Editor?
 - What is Python?
 - What is pyenv?
 - What is Django?
 - What is SQLite?
-  Installing Everything You Need
 - Step 1: Open Terminal
 - Step 2: Install Homebrew (Package Manager)
 - Step 3: Install Git
 - Step 4: Install VSCode
 - Step 5: Install pyenv
 - Step 6: Install Python with pyenv
-  Your First Steps with Terminal
 - Understanding Your File System
 - Essential Commands
 - Practice Exercise
 - Terminal Tips

-  **Setting Up Your Django Project**
 - Step 1: Navigate to Your Coding Folder
 - Step 2: Clone the Template from GitHub
 - Step 3: Open Project in VSCode
 - Step 4: Explore the Project in VSCode
 - Step 5: Create a Python Virtual Environment
 - Step 6: Install Django and Essential Packages
 - Step 7: Create the Django Project
 - Step 8: Create Your First Django App (Insurance Tracker)
 - Step 9: Configure Django Settings
 - Step 10: Create the Database Schema
 - Step 11: Create a Superuser (Admin Account)
 - Step 12: Run the Development Server
-  **Building the Insurance Tracker**
 - Understanding the Data Model
 - Step 1: Define the Database Models
 - Step 2: Create and Apply Migrations
 - Step 3: Register Models in Admin Interface
 - Step 4: Add Sample Data
 - Step 5: Create a Simple Dashboard View
 - Step 6: Configure URLs
 - Step 7: Create HTML Templates
 - Step 8: Test the Dashboard
-  **Working with AI Assistants**
 - Understanding What AI Assistants Can Do
 - Choosing the Right AI Model for Django Development
 - How to Ask Effective Questions
 - Example Feature Requests for Your Insurance Tracker
 - Working Through Errors with AI
 - Iterating on Features
 - Learning from AI-Generated Code
 - Example AI Conversation for Your Next Feature

- 🎯 Next Steps
 - Immediate Next Steps (This Week)
 - Short-term Enhancements (Next Few Weeks)
 - Learning Resources
 - Common Pitfalls to Avoid
 - Your Personal Development Path
 - Questions to Ask Yourself
 - Final Encouragement
 - Getting Help
 - Stay Updated
- 📖 Appendix: Command Reference
 - Terminal Commands
 - Git Commands
 - Python/Django Commands
 - Docker Commands (Optional - For Later)
- 🎉 You Did It!
- 🖥️ When Should I Upgrade to PostgreSQL?
 - SQLite is Sufficient For
 - Upgrade to PostgreSQL When
 - How to Upgrade (When You're Ready)
 - My Recommendation
- 🤔 "Wait, Should I Use Flask Instead?"
 - The "Flask is Simpler" Myth
 - Django vs Flask for This Project
 - Django Advantages for This Project
 - When Flask Is Actually Better
 - Where Beginners Get Stuck
 - The Uncomfortable Truth
 - The Real Question
 - My Recommendation
 - The Analogy
 - The Bottom Line



Reference Sections

-  [Troubleshooting & Common Errors](#)
 - [Understanding Django Error Pages](#)
 - [Common Errors & Solutions](#)
 - [How to Debug Like a Pro](#)
-  [Basic Git Workflow](#)
 - [Your First Commits](#)
 - [Daily Git Workflow](#)
 - [Common Git Tasks](#)
 - [When Things Go Wrong](#)
-  [Environment Variables & Secrets](#)
 - [What Are Secrets?](#)
 - [Using .env Files](#)
-  [Deployment Basics](#)
 - [Free Hosting Options](#)
 - [Quick Deploy to PythonAnywhere](#)
 - [What Changes for Production](#)
-  [What to Build Next](#)
 - [Next Features for Insurance Tracker](#)
 - [Learning Path After This Guide](#)
 - [Free Learning Resources](#)
 - [Avoiding Tutorial Hell](#)
-  [How to Ask for Help Effectively](#)
 - [Creating Minimal Reproducible Examples](#)
 - [Stack Overflow Etiquette](#)
- [What I've Tried](#)
- [Environment](#)
-  [Key Takeaways](#)
 - [Core Django Concepts to Master](#)
 - [Best Practices for Beginners](#)
 - [When You're Stuck](#)
 - [Measuring Your Progress](#)

- [Final Advice](#)

Complete Beginner's Guide to Building Your First Django Project

A step-by-step guide for absolute beginners with zero coding experience

Table of Contents

1. [What We're Building](#)
 2. [Understanding the Tools](#)
 3. [Installing Everything You Need](#)
 4. [Your First Steps with Terminal](#)
 5. [Setting Up Your Django Project](#)
 6. [Building the Insurance Tracker](#)
 7. [Working with AI Assistants](#)
 8. [Next Steps](#)
-

What We're Building

Project Example: Insurance EOB Tracker

Imagine you have medical insurance and receive "Explanation of Benefits" (EOB) documents after doctor visits. These documents show:

- How much you've spent toward your deductible
- What categories expenses fall into (office visit, lab work, prescriptions, etc.)
- How much insurance paid vs. what you owe

We're going to build a web application that:

- Lets you upload or enter EOB data
- Tracks your total spending against your deductible
- Organizes expenses by category
- Shows you at a glance how much more you need to spend before insurance kicks in
- Displays charts and

summaries of your healthcare spending

By the end of this guide, you'll have a working foundation for this project and know how to continue building it with AI assistance.

Understanding the Tools

Before we start, let's understand what each tool does and why we need it.

What is a Terminal?

Think of it as: A text-based way to talk to your computer

- **Mac:** Called "Terminal" - it's already installed
- **Windows:** Called "Command Prompt" or "PowerShell"
- **What it does:** Lets you type commands to make your computer do things (create folders, run programs, install software)
- **Why we need it:** Most programming tools are controlled through text commands

Analogy: If your computer's graphical interface (clicking icons) is like ordering food at a restaurant by pointing at pictures, the terminal is like ordering by speaking directly to the chef.

What is Git?

Think of it as: A "save game" system for your code

- **What it does:** Keeps track of every change you make to your project
- **Why we need it:**
 - You can go back to any previous version if you break something
 - You can see what changed and when
 - You can collaborate with others without overwriting their work
 - It's like "Track Changes" in Microsoft Word, but much more powerful

Analogy: Like having an infinite undo button that remembers every version of your project forever.

What is a Code Editor?

Think of it as: Microsoft Word, but for code

What it does: A text editor specifically designed for writing code that: - Colors your code to make it easier to read (syntax highlighting) - Suggests completions as you type (autocomplete) - Helps you find and fix errors (linting) - Integrates with tools like Git, terminals, and AI assistants





Analogy: You could write code in Notepad, just like you could write a novel there. But a good code editor gives you spell-check, grammar suggestions, formatting tools, and much more.

Code Editors Comparison

There are many code editors. Here's an honest comparison, especially for AI-assisted Django development:

1. Visual Studio Code (VSCode) - Recommended for Beginners

Think of it as: The Toyota Camry of code editors - reliable, popular, does everything well








-  **Free and open source**
-  **Massive extension marketplace** (50,000+ extensions)
-  **Excellent AI integration** (GitHub Copilot, multiple AI extensions)
-  **Built-in terminal, Git, debugging**
-  **Huge community** (most popular editor, easy to find help)
-  **Works on Mac, Windows, Linux**
-  **Lightweight** (starts fast, doesn't slow down your computer)
-  **AI not built-in** (need extensions)

Best for: - Beginners (this guide uses VSCode) - Web development (HTML, CSS, JavaScript, Python) - Any language really (it's the Swiss Army knife of editors)

AI capabilities: - GitHub Copilot extension (autocomplete, chat) - Claude, ChatGPT, and other AI extensions available - Can have multiple AI assistants running at once

2. Cursor - Best for AI-Assisted Coding

Think of it as: VSCode's younger sibling that was raised by AI

-  **Built-in AI chat** (GPT-4, Claude integrated directly)
-  **AI that sees your entire project** (understands context)
-  **Cmd+K for inline AI edits** (AI rewrites code right where you're working)
-  **All VSCode extensions work** (it's a fork of VSCode)
-  **Tab to accept AI suggestions** (like Copilot but better)
-  **Free tier available** (plus paid plans for more features)
-  **Newer** (smaller community than VSCode)
-  **Requires internet** (AI features need connection)









Best for: - Coding with AI assistance (this is what it's built for) - Learning to code (AI explains as you go) - Rapid prototyping (AI writes boilerplate fast)

AI capabilities: - Built-in chat with GPT-4, Claude, or other models - AI can see and edit multiple files at once - Inline AI commands (highlight code, ask AI to improve it) - AI-powered debugging

Why it's amazing for beginners: - Highlight confusing code → Cmd+K → "Explain this" - Get error → Click → AI suggests fix - Want to add feature → Cmd+L → Chat with AI about implementation

3. PyCharm (by JetBrains) - Best for Pure Python Development

Think of it as: A luxury car with all the features, but heavier

-  **Django support out-of-the-box** (understands Django patterns)
-  **Excellent debugging tools** (best debugger)
-  **Smart refactoring** (rename things safely across entire project)
-  **Database tools built-in** (query databases without leaving editor)
-  **Professional edition free for students**
-  **Heavy** (uses lots of RAM, slower startup)
-  **Learning curve** (lots of features = more complex)
-  **AI integration okay** (not as good as Cursor/VSCode with Copilot)








Best for: - Professional Python/Django developers - Large projects (1000s of files) - Data science (Jupyter notebooks built-in)

AI capabilities: - GitHub Copilot plugin available - JetBrains AI Assistant (paid feature) - Less

AI-focused than Cursor

4. Sublime Text - Fastest Editor

Think of it as: A sports car - fast, minimal, powerful









-  **Extremely fast** (opens instantly, handles huge files)
-  **Clean, distraction-free interface**
-  **Multiple cursors** (edit many places at once)
-  **Works without internet**
-  **Limited AI integration** (few AI plugins)
-  **Costs \$99** (after trial period)
-  **Smaller plugin ecosystem**

Best for: - Experienced developers who value speed - Editing config files, logs - People who don't want AI help

AI capabilities: - Basic Copilot support (unofficial) - Limited compared to modern editors

5. Vim / Neovim - For Keyboard Warriors

Think of it as: A manual transmission racecar - incredibly powerful if you know how to drive it

-  **Works in terminal** (no GUI needed)
-  **Extremely customizable** (infinite configurations)
-  **Keyboard-only** (never touch mouse)
-  **Works on any system** (including servers)
-  **Growing AI support** (Copilot.vim, ChatGPT.nvim)
-  **Steep learning curve** (takes weeks to become productive)
-  **Not beginner-friendly** (need to learn modal editing)
-  **Time investment** (configuring takes time)

Best for: - Experienced developers - People who work on remote servers - Those who want ultimate keyboard efficiency

AI capabilities: - Copilot.vim plugin (GitHub Copilot) - ChatGPT.nvim (AI chat in Neovim) -

Requires manual setup




































6. Others Worth Mentioning

Atom (deprecated, don't use) - GitHub shut it down - Use VSCode instead

Notepad++ (Windows only) - Too basic for modern development - No AI support

Replit (browser-based) - Good for learning basics - Built-in AI (Ghostwriter) - Not for serious projects

Comparison Table: AI Coding Features

Feature	VSCode	Cursor	PyCharm	Sublime	Vim
Built-in AI chat	 (extension)		 (paid)		 (plugin)
GitHub Copilot					
AI sees full project					
Inline AI edits					
Multiple AI models					
Beginner friendly					
Free		 (tier)	 (community)		

My Recommendation for This Project

For absolute beginners following this guide: → **Start with VSCode** + GitHub Copilot (or free AI extensions)

Why VSCode for beginners: 1. This guide's screenshots and instructions use VSCode 2. Massive community (easy to find help) 3. Free forever 4. Great AI extensions available 5. Standard in the industry (learn once, use everywhere)

If you want the BEST AI coding experience: → **Use Cursor** from day 1

Why Cursor for AI learners: 1. AI is built-in (no setup needed) 2. Highlight code → Ask AI to explain it (perfect for learning) 3. AI sees your whole project (better suggestions) 4. Still compatible with all VSCode extensions 5. Feels like pair programming with an expert

If you're already comfortable with another editor: → **Stick with what you know** and add AI plugins

Don't use for learning: - ❌ Vim/Neovim (too much learning curve) - ❌ PyCharm (overkill for beginners) - ❌ Sublime (weak AI support) - ❌ Notepad, TextEdit, or basic editors (you'll fight your tools)

Practical Advice: Switching Editors Later

Good news: Your code works in any editor!

- Code files are just text files
- You can switch editors anytime
- Many developers use different editors for different tasks

Common pattern: 1. Start with VSCode (learn basics) 2. Try Cursor (experience AI-first coding) 3. Stick with whichever feels better 4. Maybe try PyCharm later for large projects

Bottom line: - **For this guide:** VSCode is recommended (free, popular, well-supported) - **For AI-assisted learning:** Cursor is amazing (AI built-in, still based on VSCode) - **Pick one and stick with it for 3 months** before switching (avoid tool-hopping)

The editor doesn't make you a better programmer - practice does. But a good editor with AI assistance will make learning faster and more enjoyable!

What is Python?

Think of it as: The language we'll write in

- **What it does:** A programming language that's easy for humans to read and write
- **Why we need it:** Django (our web framework) is built with Python
- **Why Python:** It reads almost like English, making it perfect for beginners

Analogy: If building a website is like building a house, Python is the language you speak to tell the construction workers what to do.

What is pyenv?

Think of it as: A manager for different versions of Python

- **What it does:** Lets you have multiple versions of Python on your computer
- **Why we need it:** Different projects might need different Python versions; this keeps them separate
- **Bonus:** Creates "virtual environments" - isolated workspaces for each project

Analogy: Like having different toolboxes for different jobs. Your plumbing toolbox doesn't mix with your electrical toolbox.

What is Django?

Think of it as: A website construction kit

- **What it does:** Provides pre-built components for common website features (user login, database, admin panel, etc.)
- **Why we need it:** Instead of building everything from scratch, we get a professional foundation
- **What's included:**
 - Database system (stores your data)
 - User authentication (login/logout)
 - Admin interface (manage your data)
 - Security features (keeps hackers out)
 - URL routing (connects web addresses to pages)

Analogy: Building a website from scratch is like building a car from raw metal. Django is like getting a car chassis, engine, and wheels already assembled - you just customize it.

What is SQLite?

Think of it as: A smart filing cabinet in a single file

- **What it does:** Stores all your application's data in organized tables
- **Why we're using it:**
 - Built into Python (no installation needed!)
 - Perfect for learning and personal projects
 - Just a single file you can see and backup easily
 - Fast enough for thousands of records
- **What it stores:** Your EOB entries, categories, spending amounts, user accounts, etc.

Analogy: You could keep all your data in separate text files scattered around, or you could use a self-contained filing system that keeps everything organized in one place. SQLite is that filing system.

For beginners: SQLite is perfect. When your app grows to thousands of users, you can easily upgrade to PostgreSQL later (takes 10 minutes). But for now, one less thing to install and manage!



Installing Everything You Need

Let's install each tool step by step. These instructions are for **Mac** (since you're on macOS). Windows users will have slightly different steps.

Step 1: Open Terminal

1. Press `Command + Space` (opens Spotlight search)
2. Type "Terminal"
3. Press Enter

You'll see a window with white or black background and a blinking cursor. This is your terminal.

What you'll see: Something like:

```
YourName@MacBook-Pro ~ %
```

This is called a "prompt" - it's waiting for you to type a command.

Step 2: Install Homebrew (Package Manager)

What is Homebrew? Think of it as an "App Store for developers" - it installs programming tools easily.

1. In Terminal, paste this command (copy everything on one line):

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Press Enter
3. Enter your Mac password when prompted (you won't see characters as you type - this is normal)
4. Wait 5-10 minutes for installation

How to verify it worked:

```
brew --version
```

You should see something like: `Homebrew 4.x.x`

Step 3: Install Git

In Terminal, type:

```
brew install git
```

Wait for it to finish, then verify:

```
git --version
```

You should see: `git version 2.x.x`

Configure Git with your name and email:

```
git config --global user.name "Your Name"  
git config --global user.email "your.email@example.com"
```

Replace "Your Name" and the email with your actual information.

Step 4: Install VSCode

Two ways to install:

Option A: Download from website 1. Go to <https://code.visualstudio.com/> 2. Click the download button for Mac 3. Open the downloaded file 4. Drag "Visual Studio Code" to your Applications folder

Option B: Install with Homebrew

```
brew install --cask visual-studio-code
```

Verify: Open VSCode by pressing `Command + Space`, typing "Visual Studio Code", and pressing Enter.

Step 5: Install pyenv

In Terminal:

```
brew install pyenv
```

Configure your shell to use pyenv:

1. Check what shell you're using:

```
echo $SHELL
```

If it says `/bin/zsh`, you're using zsh (most modern Macs).

2. Add pyenv to your shell configuration:


```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc
echo 'eval "$(pyenv init --path)"' >> ~/.zshrc
echo 'eval "$(pyenv init -)"' >> ~/.zshrc
```

3. Restart your terminal (close and reopen it) or run:

```
source ~/.zshrc
```

Verify:

```
pyenv --version
```

Step 6: Install Python with pyenv

Install Python 3.13.3 (or latest version):

```
pyenv install 3.13.3
```

This takes 5-10 minutes.

Set it as your global Python version:

```
pyenv global 3.13.3
```

Verify:

```
python --version
```

Should show: Python 3.13.3

That's it for installation! 

You now have everything you need to build Django applications. We're using SQLite for the database (built into Python), so no database installation needed.

Your First Steps with Terminal

Before we dive into the project, let's learn basic terminal commands. Think of these as learning to walk before you run.

Understanding Your File System

Key concept: Everything on your computer is organized in folders (also called "directories"). The terminal lets you navigate these folders using text commands.

The filesystem tree:

```
/                                (root - the top of everything)
├── Users
│   ├── YourName                (your home directory)
│   │   ├── Desktop
│   │   ├── Documents
│   │   ├── Downloads
│   │   └── Coding              (we'll create this)
```

Essential Commands

Open Terminal and try these:

1. `pwd` - Print Working Directory

Shows where you are right now

```
pwd
```

Output: `/Users/YourName` (your current location)

2. `ls` - List

Shows files and folders in current location

```
ls
```

Output: `Desktop Documents Downloads Pictures`

See more details:

```
ls -la
```

This shows hidden files, permissions, and sizes.

3. `cd` - Change Directory

Move to a different folder

```
cd Desktop          # Go to Desktop
cd ..               # Go up one level (to parent folder)
cd ~                # Go to home directory
cd /Users/YourName/Documents # Go to specific path
```

Practice:

```
cd Desktop
pwd                # Should show /Users/YourName/Desktop
cd ..
pwd                # Should show /Users/YourName
```

4. `mkdir` - Make Directory

Create a new folder

```
mkdir Coding
```

This creates a "Coding" folder in your current location.

5. `cp` - Copy

Copy files or folders

```
cp file.txt backup.txt          # Copy a file
cp -r folder backup_folder      # Copy a folder (-r means recursive)
```

6. `mv` - Move or Rename

Move or rename files/folders

```
mv oldname.txt newname.txt      # Rename
mv file.txt Documents/          # Move to Documents folder
```

7. `rm` - Remove

Delete files or folders

```
rm file.txt                    # Delete a file
rm -r folder                   # Delete a folder (-r means recursive)
```

 **WARNING:** `rm` is permanent - there's no Trash/Recycle Bin!

Practice Exercise

Let's create a practice project structure:

```
# Go to your home directory
cd ~

# Create a Coding folder
mkdir Coding

# Go into it
cd Coding

# Create a practice folder
mkdir practice

# Go into it
cd practice

# Create some files (the 'echo' command creates a file with text)
echo "Hello World" > test.txt

# See what's in the file
cat test.txt

# Create a folder structure
mkdir -p project/src/components

# See the structure
ls -R
```

What each command did: - `mkdir -p` creates nested folders (creates all parents if needed) -
`echo "text" > file.txt` creates a file with that text - `cat file.txt` shows the contents

of a file - `ls -R` lists everything recursively (shows subfolders too)

Terminal Tips

Shortcuts to save time: - `Tab` - Auto-complete (start typing a filename and press Tab) - `↑` (Up Arrow) - Shows previous command - `Ctrl + C` - Stop a running command - `Ctrl + A` - Go to beginning of line - `Ctrl + E` - Go to end of line - `Ctrl + U` - Clear entire line - `clear` - Clear the terminal screen

Practice auto-complete:

```
cd ~/Cod<TAB>      # Press Tab after typing 'Cod', it will complete to  
                  'Coding'
```



Setting Up Your Django Project

Now we'll set up the actual project that will become your insurance tracker.

Step 1: Navigate to Your Coding Folder

```
cd ~/Coding  
pwd      # Verify you're in /Users/YourName/Coding
```

Step 2: Clone the Template from GitHub

We'll download (clone) the Django template from GitHub:

```
# Clone the template repository  
git clone https://github.com/cloudflying87/Django_template_build.git  
    InsuranceTracker  
  
# Go into your new project  
cd InsuranceTracker
```

What just happened: - `git clone` downloaded the entire template from GitHub - We named our local copy "InsuranceTracker" - The folder contains all the starter files we need

What is cloning? Think of it like downloading a zip file, but better: - You get the entire project with all its history - You can pull updates if the template improves - Git tracks everything automatically

For advanced users with SSH keys: If you have SSH keys set up with GitHub, you can use:

```
git clone git@github.com:cloudflying87/Django_template_build.git
InsuranceTracker
```

Troubleshooting: - If you get "command not found: git", go back to Step 3 in the installation section - If you get a permission error, make sure you're in a folder where you have write access (like ~/Coding)

Step 3: Open Project in VSCode

```
code .
```

What this does: Opens the current folder (`.` means "current folder") in VSCode.

First time only: If VSCode doesn't open, you may need to install the `code` command: 1. Open VSCode manually 2. Press `Command + Shift + P` 3. Type "shell command" 4. Click "Shell Command: Install 'code' command in PATH" 5. Try `code .` again

Step 4: Explore the Project in VSCode

In VSCode, you'll see a sidebar with folders. This is your project structure. Let me explain what each folder does:

```

InsuranceTracker/
├─ apps/                # Where your Django apps live (we'll create
'insurance' here)
├─ build/               # Build scripts for deployment (advanced - ignore
for now)
├─ config/              # Django project settings (will be created)
├─ docs/                # Documentation
├─ static/              # CSS, JavaScript, images
├─   └─ css/            # Your custom styles
├─   └─ js/              # JavaScript files
├─   └─ images/          # Images for your site
├─ templates/           # HTML templates for your pages
├─ .env.example         # Example environment variables
├─ requirements.txt     # Python packages needed (will be created)
├─ db.sqlite3           # Your database file (will be created)
└─ manage.py            # Django's command-line tool (will be created)

```

Don't worry if some of these don't exist yet! We'll create them as we go. The template provides the basic structure.

Step 5: Create a Python Virtual Environment

What is a virtual environment? Think of it as a separate workspace for this project's tools. Just like you might have separate toolboxes for different hobbies (one for woodworking, one for electronics), a virtual environment keeps this project's Python packages separate from other projects.

Why do we need this? - Project A might need Django version 4.0 - Project B might need Django version 5.0 - Without virtual environments, you'd have constant conflicts - With virtual environments, each project has its own isolated set of packages

Let's create one:

```

# Make sure you're in the project folder
cd ~/Coding/InsuranceTracker

# Create a virtual environment named 'insurance-tracker' using Python 3.13.3
pyenv virtualenv 3.13.3 insurance-tracker

# Activate it for this folder
pyenv local insurance-tracker

```

What each command does:

1. **pyenv virtualenv 3.13.3 insurance-tracker**

- Creates a new virtual environment
- Uses Python version 3.13.3
- Names it "insurance-tracker"
- This is like creating a new toolbox

2. `pyenv local insurance-tracker`

- Creates a `.python-version` file in your folder
- This file tells pyenv "whenever I'm in this folder, use the insurance-tracker environment"
- It's automatic from now on - every time you `cd` into this folder, the right environment activates

Verify it's active:

```
python --version    # Should show Python 3.13.3
which python        # Should show a path with 'insurance-tracker' in it
```

What you should see:

```
Python 3.13.3
/Users/YourName/.pyenv/versions/insurance-tracker/bin/python
```

The path shows "insurance-tracker" - that means you're using the right environment!

Troubleshooting: - If `python --version` shows a different version, try closing and reopening your terminal - Make sure you're in the InsuranceTracker folder (`pwd` to check) - If it still doesn't work, run `pyenv local insurance-tracker` again

Step 6: Install Django and Essential Packages

First, let's create a `requirements.txt` file that lists what packages we need:

In VSCode: 1. Click "New File" icon in the left sidebar (or press `Command + N`) 2. Paste this content:

```
Django==5.0.1
python-dotenv==1.0.0
pillow==10.2.0
```

3. Save as `requirements.txt` in the root folder (File → Save, or `Command + S`)

What each package does:

- **Django==5.0.1** - The web framework itself
 - This is the main package that provides everything (models, views, templates, admin, etc.)
 - The `==5.0.1` means "install exactly version 5.0.1" (ensures consistency)
- **python-dotenv==1.0.0** - Loads environment variables from `.env` file
 - Lets you store secrets (passwords, API keys) in a file that's not committed to Git
 - Example: instead of hardcoding your email password in code, you store it in `.env`
- **pillow==10.2.0** - Handles image uploads
 - Useful for uploading photos of receipts or EOB documents
 - Django needs this for ImageField and processing images

Notice what we DON'T need: - No PostgreSQL driver (we're using SQLite, which is built into Python) - No Redis or Celery (not needed for a simple tracker) - Just 3 packages! Keeping it simple.

Install these packages:

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


What this does: - `pip` is Python's package installer (like an app store) - `-r requirements.txt` means "read the list from this file" - It downloads and installs each package and their dependencies

This takes 1-2 minutes. You'll see lots of output - that's normal!

Verify Django is installed:

```
django-admin --version
```

Should show: `5.0.1`

If you see `5.0.1`, you're good to go! 

Step 7: Create the Django Project

```
# Create the project configuration
django-admin startproject config .
```

Important: Note the `.` at the end! This means “create the project in the current folder” rather than creating a new subfolder.

What this created:

```
config/
├─ __init__.py      # Makes this a Python package
├─ settings.py      # Main Django settings
├─ urls.py          # URL routing configuration
├─ asgi.py          # Async server config
├─ wsgi.py          # Web server config
└─ manage.py        # Django management command tool
```

Verify it worked:

```
python manage.py --version
```

Should show: `5.0.1`

Step 8: Create Your First Django App (Insurance Tracker)

Django projects are organized into “apps”. Each app handles a specific feature. We’ll create an app for our insurance tracking.

```
# Create the app
python manage.py startapp insurance

# Move it into the apps/ folder (better organization)
mv insurance apps/
```

What this created:

```
apps/insurance/  
├─ __init__.py  
├─ admin.py           # Register models for admin interface  
├─ apps.py           # App configuration  
├─ models.py         # Database models (we'll define EOB data here)  
├─ tests.py          # Write tests here  
├─ views.py          # Handle web requests  
└─ migrations/       # Database change history
```

Step 9: Configure Django Settings

We need to tell Django about our new app and configure file storage for uploads.

Open `config/settings.py` in VSCode.

This file contains all of Django's configuration. Let's make a few changes:

A. Register Your Insurance App

Find the `INSTALLED_APPS` list (around line 33) and add your app:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'apps.insurance',           # Add this line  
]
```

What this does: - Tells Django "this app exists and should be included" - Django will now look for models, templates, and views in this app - The admin interface will be able to find your models

Why `apps.insurance` ? - Because we moved the `insurance` folder into `apps/` - The dot notation means "inside the apps folder, find insurance" - If we hadn't moved it, we'd just write `'insurance'`

B. Check the Database Configuration

Scroll down to the `DATABASES` section (around line 75).

You should see something like this:

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

Don't change this! This is Django's default, and it's perfect for learning.

What this means: - **ENGINE** - Which database system to use (SQLite in this case) - **NAME** - Where to store the database file - **BASE_DIR** is your project root folder - **/** joins paths (works on Mac, Windows, Linux) - This creates a file called **db.sqlite3** in your project root

Why SQLite is great for beginners: - No installation needed (built into Python) - Just a single file - easy to understand - Easy to delete and start over (just delete the file) - Fast enough for thousands of records - When you're ready, switching to PostgreSQL is just changing these 4 lines

C. Configure Static and Media Files

Find **STATIC_URL** (around line 117) and add these lines after it:

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = [  
    BASE_DIR / 'static',  
]  
STATIC_ROOT = BASE_DIR / 'staticfiles'  
  
MEDIA_URL = '/media/'  
MEDIA_ROOT = BASE_DIR / 'media'
```

What these settings do:

- **STATIC_URL = '/static/'**
 - The URL prefix for static files (CSS, JavaScript, images that are part of your site design)
 - Example: A CSS file at `static/css/style.css` will be accessible at `http://localhost:8000/static/css/style.css`
- **STATICFILES_DIRS**
 - List of folders where Django should look for static files during development
 - We're telling it to look in the `static/` folder in our project root
- **STATIC_ROOT**

- Where to collect ALL static files when deploying to production
- The `python manage.py collectstatic` command copies everything here
- Not used during development
- **MEDIA_URL = '/media/'**
 - The URL prefix for user-uploaded files (EOB documents, photos, etc.)
 - Example: An uploaded file will be accessible at
`http://localhost:8000/media/eob_documents/file.pdf`
- **MEDIA_ROOT**
 - Where to actually store uploaded files on your hard drive
 - Creates a `media/` folder in your project root

The difference between STATIC and MEDIA: - **STATIC** = Files you create (your CSS, your JavaScript, your logo) - **MEDIA** = Files users upload (EOB PDFs, receipt photos)

Save the file (`Command + S`)

Your Django settings are now configured! 

Step 10: Create the Database Schema

Now we'll create the database tables Django needs:

```
# Create migrations (Django analyzes your models and creates database
instructions)
python manage.py makemigrations

# Apply migrations (Actually creates the database tables)
python manage.py migrate
```

What you'll see:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  ...
```

This created tables for user authentication, sessions, and admin interface.

Step 11: Create a Superuser (Admin Account)

```
python manage.py createsuperuser
```

You'll be prompted for: - Username: (enter anything, like `admin`) - Email: (enter your email or press Enter to skip) - Password: (enter a password - you won't see it as you type) - Password (again): (confirm)

Remember these credentials! You'll use them to log into the admin interface.

Step 12: Run the Development Server

```
python manage.py runserver
```

What you'll see:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
January 01, 2025 - 12:00:00
Django version 5.0.1, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Open your browser and go to: `http://127.0.0.1:8000/`

You should see the Django welcome page! 🎉

Try the admin interface: `http://127.0.0.1:8000/admin/`

Log in with your superuser credentials.

To stop the server: Press `Ctrl + C` in the terminal.



Building the Insurance Tracker

Now let's build the actual insurance tracking functionality!

Understanding the Data Model

Before we code, let's think about what data we need to track:

For the Insurance Policy: - Deductible amount (e.g., \$2,000) - Deductible period (annual, per-incident, etc.) - Current year - Policy holder name

For Each EOB Entry: - Date of service - Provider name (doctor, hospital, etc.) - Service category (office visit, lab work, prescription, etc.) - Billed amount (what provider charged) - Insurance paid amount - You owe amount - Applied to deductible amount - Status (pending, paid, etc.) - Notes - Uploaded EOB document (PDF or image)

Step 1: Define the Database Models

Open `apps/insurance/models.py` in VSCode.

Replace the entire file with:

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

class InsurancePolicy(models.Model):
    """Represents an insurance policy with deductible tracking"""

    user = models.ForeignKey(User, on_delete=models.CASCADE)
    policy_name = models.CharField(max_length=200, help_text="e.g., 'Blue Cross 2025'")
    deductible_amount = models.DecimalField(max_digits=10, decimal_places=2, help_text="Annual deductible")
    deductible_period = models.CharField(
        max_length=50,
        choices=[
            ('annual', 'Annual'),
            ('per_incident', 'Per Incident'),
            ('lifetime', 'Lifetime'),
        ],
        default='annual'
    )
    year = models.IntegerField(default=timezone.now().year)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-year', '-created_at']

    def __str__(self):
```

```

        return f"{self.policy_name} - {self.year}"

    def total_spent(self):
        """Calculate total amount applied to deductible"""
        return self.eob_entries.aggregate(
            total=models.Sum('applied_to_deductible')
        )['total'] or 0

    def remaining_deductible(self):
        """Calculate remaining deductible amount"""
        return max(0, self.deductible_amount - self.total_spent())

    def deductible_met_percentage(self):
        """Calculate percentage of deductible met"""
        if self.deductible_amount == 0:
            return 0
        return min(100, (self.total_spent() / self.deductible_amount) * 100)

class ServiceCategory(models.Model):
    """Categories for medical services"""

    name = models.CharField(max_length=100, unique=True)
    description = models.TextField(blank=True)
    color = models.CharField(max_length=7, default='#3B82F6', help_text="Hex color for charts")

    class Meta:
        verbose_name_plural = 'Service Categories'
        ordering = ['name']

    def __str__(self):
        return self.name

class EOBEEntry(models.Model):
    """Explanation of Benefits entry"""

    policy = models.ForeignKey(InsurancePolicy, on_delete=models.CASCADE,
                              related_name='eob_entries')

    # Service details
    date_of_service = models.DateField()
    provider_name = models.CharField(max_length=200)
    category = models.ForeignKey(ServiceCategory, on_delete=models.SET_NULL,
                                 null=True, blank=True)
    service_description = models.TextField(blank=True)

    # Financial details
    billed_amount = models.DecimalField(max_digits=10, decimal_places=2,
                                         help_text="Amount provider charged")
    insurance_paid = models.DecimalField(max_digits=10, decimal_places=2,
                                         help_text="Amount insurance paid")

```



```

patient_owed = models.DecimalField(max_digits=10, decimal_places=2,
    help_text="Amount you owe")
applied_to_deductible = models.DecimalField(max_digits=10,
    decimal_places=2, help_text="Amount applied to deductible")

# Status and tracking
status = models.CharField(
    max_length=20,
    choices=[
        ('pending', 'Pending'),
        ('paid', 'Paid'),
        ('partially_paid', 'Partially Paid'),
        ('disputed', 'Disputed'),
    ],
    default='pending'
)

payment_date = models.DateField(null=True, blank=True)
notes = models.TextField(blank=True)

# File upload
eob_document = models.FileField(upload_to='eob_documents/', null=True,
    blank=True)

# Timestamps
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)

class Meta:
    verbose_name = 'EOB Entry'
    verbose_name_plural = 'EOB Entries'
    ordering = ['-date_of_service', '-created_at']

def __str__(self):
    return f"{self.provider_name} - {self.date_of_service} "
    ($ {self.patient_owed})"

```

What this code does:

1. **InsurancePolicy model:** Stores policy information and calculates deductible progress
 - `total_spent()` - Adds up all EOB entries
 - `remaining_deductible()` - Shows how much left to pay
 - `deductible_met_percentage()` - For progress bars
2. **ServiceCategory model:** Categories like "Office Visit", "Lab Work", etc.
3. **EOBEntry model:** Each individual EOB with all financial details

Save the file (`Command + S`)

Step 2: Create and Apply Migrations

```
# Create migration files based on our new models
python manage.py makemigrations insurance

# Apply the migrations to create database tables
python manage.py migrate insurance
```

What you'll see:

```
Migrations for 'insurance':
  apps/insurance/migrations/0001_initial.py
    - Create model InsurancePolicy
    - Create model ServiceCategory
    - Create model EOEntry
```

Step 3: Register Models in Admin Interface

Open `apps/insurance/admin.py` in VSCode.

Replace the entire file with:

```
from django.contrib import admin
from .models import InsurancePolicy, ServiceCategory, EOEntry

@admin.register(ServiceCategory)
class ServiceCategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'description', 'color']
    search_fields = ['name']

@admin.register(InsurancePolicy)
class InsurancePolicyAdmin(admin.ModelAdmin):
    list_display = ['policy_name', 'year', 'deductible_amount',
                    'get_total_spent', 'get_remaining']
    list_filter = ['year']
    search_fields = ['policy_name', 'user__username']

    def get_total_spent(self, obj):
        return f"${obj.total_spent():.2f}"
    get_total_spent.short_description = 'Total Spent'

    def get_remaining(self, obj):
        return f"${obj.remaining_deductible():.2f}"
    get_remaining.short_description = 'Remaining'
```

```
@admin.register(EOBEntry)
class EOBEntryAdmin(admin.ModelAdmin):
    list_display = [
        'date_of_service',
        'provider_name',
        'category',
        'patient_owed',
        'applied_to_deductible',
        'status'
    ]
    list_filter = ['status', 'category', 'date_of_service']
    search_fields = ['provider_name', 'service_description']
    date_hierarchy = 'date_of_service'

    fieldsets = (
        ('Service Information', {
            'fields': ('policy', 'date_of_service', 'provider_name',
                       'category', 'service_description')
        }),
        ('Financial Details', {
            'fields': ('billed_amount', 'insurance_paid', 'patient_owed',
                       'applied_to_deductible')
        }),
        ('Status & Payment', {
            'fields': ('status', 'payment_date', 'notes')
        }),
        ('Documents', {
            'fields': ('eob_document',)
        }),
    )
)
```

What this does: - Registers our models so they appear in the admin interface - Customizes how data is displayed - Adds filters and search functionality - Organizes fields into logical sections

Save the file (Command + S)

Step 4: Add Sample Data

Let's add some sample categories and a test policy.

Start the development server:

```
python manage.py runserver
```

Go to the admin interface: <http://127.0.0.1:8000/admin/>

Log in with your superuser credentials.

Add Service Categories: 1. Click "Service Categories" → "Add Service Category" 2. Add these categories one by one: - Name: "Office Visit", Color: #3B82F6 (blue) - Name: "Lab Work", Color: #10B981 (green) - Name: "Prescription", Color: #F59E0B (amber) - Name: "Imaging (X-Ray, MRI)", Color: #8B5CF6 (purple) - Name: "Emergency Room", Color: #EF4444 (red) - Name: "Surgery", Color: #EC4899 (pink) - Name: "Physical Therapy", Color: #06B6D4 (cyan)

Add an Insurance Policy: 1. Click "Insurance Policies" → "Add Insurance Policy" 2. Fill in: - User: Select your username - Policy name: "My Health Insurance 2025" - Deductible amount: 2000 - Deductible period: Annual - Year: 2025 3. Click "Save"

Add Sample EOB Entries: 1. Click "EOB Entries" → "Add EOB Entry" 2. Example 1: - Policy: Select the policy you just created - Date of service: Pick a recent date - Provider name: "Dr. Smith - Family Practice" - Category: Office Visit - Service description: "Annual checkup" - Billed amount: 250 - Insurance paid: 200 - Patient owed: 50 - Applied to deductible: 50 - Status: Paid 3. Click "Save and add another" 4. Add a few more entries with different categories and amounts

Step 5: Create a Simple Dashboard View

Now let's create a web page to view this data (instead of just using the admin).

Open `apps/insurance/views.py` in VSCode.

Replace with:

```
from django.shortcuts import render, get_object_or_404
from django.contrib.auth.decorators import login_required
from django.db.models import Sum
from .models import InsurancePolicy, EOEntry, ServiceCategory

@login_required
def dashboard(request):
    """Main dashboard showing all policies"""
    policies = InsurancePolicy.objects.filter(user=request.user)

    context = {
        'policies': policies,
    }
    return render(request, 'insurance/dashboard.html', context)

@login_required
def policy_detail(request, policy_id):
    """Detailed view of a single policy with all EOB entries"""
    policy = get_object_or_404(InsurancePolicy, id=policy_id,
                               user=request.user)

    # Get all EOB entries for this policy
    eob_entries = policy.eob_entries.all()

    # Calculate spending by category
    category_spending = eob_entries.values('category__name',
                                           'category__color').annotate(
        total=Sum('applied_to_deductible')
    ).order_by('-total')

    context = {
        'policy': policy,
        'eob_entries': eob_entries,
        'category_spending': category_spending,
        'total_entries': eob_entries.count(),
    }
    return render(request, 'insurance/policy_detail.html', context)
```

Step 6: Configure URLs

Create a new file: `apps/insurance/urls.py`

```

from django.urls import path
from . import views

app_name = 'insurance'

urlpatterns = [
    path('', views.dashboard, name='dashboard'),
    path('policy/<int:policy_id>/', views.policy_detail,
         name='policy_detail'),
]

```

Open `config/urls.py` and modify it to include our app's URLs:

```

from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.insurance.urls')), # Add this line
]

# Serve media files in development
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)

```

Step 7: Create HTML Templates

Create the template directories:

```
mkdir -p templates/insurance
```

Create `templates/base.html` :

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Insurance Tracker{% endblock %}</title>
    <style>
        * {
            margin: 0;

```

```
padding: 0;
box-sizing: border-box;
}

body {
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI',
  sans-serif;
  line-height: 1.6;
  color: #333;
  background: #f5f5f5;
}

.container {
  max-width: 1200px;
  margin: 0 auto;
  padding: 20px;
}

header {
  background: #3B82F6;
  color: white;
  padding: 20px 0;
  margin-bottom: 30px;
}

header h1 {
  font-size: 24px;
  font-weight: 600;
}

.card {
  background: white;
  border-radius: 8px;
  padding: 20px;
  margin-bottom: 20px;
  box-shadow: 0 1px 3px rgba(0,0,0,0.1);
}

.progress-bar {
  background: #e5e7eb;
  border-radius: 9999px;
  height: 24px;
  overflow: hidden;
  margin: 10px 0;
}

.progress-fill {
  background: #3B82F6;
  height: 100%;
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
```

```
        font-size: 12px;
        font-weight: 600;
        transition: width 0.3s;
    }

    table {
        width: 100%;
        border-collapse: collapse;
    }

    th, td {
        padding: 12px;
        text-align: left;
        border-bottom: 1px solid #e5e7eb;
    }

    th {
        background: #f9fafb;
        font-weight: 600;
    }

    .status-badge {
        display: inline-block;
        padding: 4px 12px;
        border-radius: 9999px;
        font-size: 12px;
        font-weight: 600;
    }

    .status-paid {
        background: #d1fae5;
        color: #065f46;
    }

    .status-pending {
        background: #fee2e2;
        color: #991b1b;
    }

    .btn {
        display: inline-block;
        padding: 10px 20px;
        background: #3b82f6;
        color: white;
        text-decoration: none;
        border-radius: 6px;
        font-weight: 500;
    }

    .btn:hover {
        background: #2563eb;
    }
</style>
```



```
</head>
<body>
  <header>
    <div class="container">
      <h1>🚗 Insurance Tracker</h1>
    </div>
  </header>

  <div class="container">
    {% block content %}{% endblock %}
  </div>
</body>
</html>
```

Create `templates/insurance/dashboard.html` :

```
{% extends 'base.html' %}

{% block title %}Dashboard - Insurance Tracker{% endblock %}

{% block content %}
<h2>Your Insurance Policies</h2>

{% if policies %}
    {% for policy in policies %}
        <div class="card">
            <h3>{{ policy.policy_name }}</h3>
            <p><strong>Deductible:</strong> ${{
                policy.deductible_amount|floatformat:2 }}</p>

            <div class="progress-bar">
                <div class="progress-fill" style="width: {{
                    policy.deductible_met_percentage }}%">
                    {{ policy.deductible_met_percentage|floatformat:0 }}%
                </div>
            </div>

            <p>
                <strong>Spent:</strong> ${{ policy.total_spent|floatformat:2 }}
                |
                <strong>Remaining:</strong> ${{
                    policy.remaining_deductible|floatformat:2 }}
            </p>

            <a href="{% url 'insurance:policy_detail' policy.id %}"
                class="btn">View Details</a>
        </div>
    {% endfor %}
{% else %}
    <div class="card">
        <p>No insurance policies found. <a
            href="/admin/insurance/insurancepolicy/add/">Add one in the admin</a>
        </p>
    </div>
{% endif %}
{% endblock %}
```

Create templates/insurance/policy_detail.html :

```
{% extends 'base.html' %}

{% block title %}{{ policy.policy_name }} - Insurance Tracker{% endblock %}

{% block content %}
<a href="{% url 'insurance:dashboard' %}">&larr; Back to Dashboard</a>
```

```

<div class="card">
  <h2>{{ policy.policy_name }}</h2>

  <div class="progress-bar">
    <div class="progress-fill" style="width: {{
      policy.deductible_met_percentage }}%">
      {{ policy.deductible_met_percentage|floatformat:0 }}%
    </div>
  </div>

  <p>
    <strong>Total Deductible:</strong> ${{
      policy.deductible_amount|floatformat:2 }}<br>
    <strong>Amount Spent:</strong> ${{ policy.total_spent|floatformat:2
      }}<br>
    <strong>Remaining:</strong> ${{
      policy.remaining_deductible|floatformat:2 }}
  </p>
</div>

<div class="card">
  <h3>Spending by Category</h3>
  {% if category_spending %}
    {% for cat in category_spending %}
      <div style="margin-bottom: 10px;">
        <div style="display: flex; justify-content: space-between;
          margin-bottom: 4px;">
          <span>{{ cat.category__name }}</span>
          <strong>${{ cat.total|floatformat:2 }}</strong>
        </div>
        <div class="progress-bar" style="height: 8px;">
          <div class="progress-fill"
            style="width: {{ cat.total|floatformat:0 }}%;
            background: {{ cat.category__color }}">
          </div>
        </div>
      </div>
    {% endfor %}
  {% else %}
    <p>No spending data yet.</p>
  {% endif %}
</div>

<div class="card">
  <h3>EOB Entries ({{ total_entries }})</h3>

  {% if eob_entries %}
    <table>
      <thead>
        <tr>
          <th>Date</th>
          <th>Provider</th>
          <th>Category</th>
        </tr>
      </thead>
    </table>
  {% endif %}

```

```

        <th>Billed</th>
        <th>Insurance Paid</th>
        <th>You Owe</th>
        <th>To Deductible</th>
        <th>Status</th>
    </tr>
</thead>
<tbody>
    {% for entry in eob_entries %}
    <tr>
        <td>{{ entry.date_of_service }}</td>
        <td>{{ entry.provider_name }}</td>
        <td>{{ entry.category.name|default:"-"}</td>
        <td>${{ entry.billed_amount|floatformat:2 }}</td>
        <td>${{ entry.insurance_paid|floatformat:2 }}</td>
        <td>${{ entry.patient_owed|floatformat:2 }}</td>
        <td>${{ entry.applied_to_deductible|floatformat:2 }}</td>
        <td>
            <span class="status-badge status-{{ entry.status }}">
                {{ entry.get_status_display }}
            </span>
        </td>
    </tr>
    {% endfor %}
</tbody>
</table>
{% else %}
<p>No EOB entries yet. <a href="/admin/insurance/eobentry/add/">Add one
    in the admin</a></p>
{% endif %}
</div>
{% endblock %}

```

Step 8: Test the Dashboard

Make sure your server is running:

```
python manage.py runserver
```

Visit: <http://127.0.0.1:8000/>

You should see your insurance policy with a progress bar!

Click “View Details” to see the full breakdown with all EOB entries and category spending.



Working with AI Assistants

Now that you have a working foundation, here's how to use AI assistants (like Claude, ChatGPT, etc.) to build features faster.

Understanding What AI Assistants Can Do

AI assistants excel at: - Writing boilerplate code - Explaining existing code - Debugging errors - Suggesting implementations - Creating HTML/CSS layouts - Writing database queries - Generating test data

AI assistants struggle with: - Understanding your specific business logic (you need to explain it) - Knowing your exact file structure (you need to show them) - Making subjective design decisions (you need to decide) - Complex debugging across multiple files

Choosing the Right AI Model for Django Development

Not all AI models are created equal for coding tasks. Here's an honest comparison to help you choose:

Understanding Context Windows and Tokens

Before we compare models, let's understand what these technical terms mean:

What is a token? - A "token" is roughly a word, part of a word, or a piece of code - Examples: - "Hello" = 1 token - "Django" = 1 token - "def calculate_total():" = about 6 tokens - Your entire `models.py` file = maybe 1,000-3,000 tokens

Rule of thumb: - 1 token \approx 4 characters of text - 100 tokens \approx 75 words - 1,000 tokens \approx 750 words or \sim 50 lines of code

What is a context window? Think of it as the AI's "working memory" - how much text it can see and remember at once.

Analogy: Imagine you're editing a book: - **Small context (4K tokens):** You can only see 1 page at a time - **Medium context (16K tokens):** You can see 1 chapter - **Large context (128K tokens):** You can see 5-10 chapters - **Huge context (200K tokens):** You can see the entire book plus notes

Why this matters for Django:

Example 1: Debugging an error

Small context (GPT-3.5 - 4K tokens):

- Can see: Just the error message and one function
- Problem: Doesn't know what's in your `models.py` or `settings.py`
- Result: Generic advice that might not apply

Large context (Claude - 200K tokens):

- Can see: Error + `models.py` + `views.py` + `settings.py` + `templates`
- Understands: Full picture of your app
- Result: Specific fix that considers your entire setup

Example 2: Your Insurance Tracker

Your project files: - `models.py` (EOB models) = ~2,000 tokens - `views.py` = ~1,500 tokens - `admin.py` = ~800 tokens - `settings.py` = ~1,000 tokens - `urls.py` = ~500 tokens -
Total: ~6,000 tokens

With small context (4K tokens): - You can only show the AI one file at a time - AI doesn't know how files connect - You have to explain relationships manually

With large context (200K tokens): - Show AI your entire project at once - AI understands how `models`, `views`, and `templates` work together - Get better, more integrated suggestions

Practical example:





You ask: "Why is my `EOBEntry` not saving the category?"




Small context AI: - Sees only the model definition - Says: "The code looks fine, make sure you're setting the category field" - Doesn't help much

Large context AI: - Sees `models.py`, `views.py`, and your form - Says: "In your `policy_detail` view on line 23, you're creating the `EOBEntry` but not including the category in the form data. Add `category=request.POST.get('category')` to line 25." - Actually solves the problem!








The bottom line: - **Bigger context = AI sees more of your code = better answers** - For learning Django, aim for at least 100K token context - This is why Claude (200K) and GPT-4 (128K) are recommended - Avoid older models with tiny context windows (you'll spend time explaining instead of building)

Best for Django/Python Projects





1. **Claude (Anthropic) - Sonnet 3.5 or Opus** -  **Best for:** Complex Django projects, debugging, refactoring -  **Context window:** 200K tokens (can see your entire codebase) -  **Code quality:** Excellent at following Django best practices -  **Explaining code:** Great at

breaking down complex concepts for beginners -  **Multi-file changes:** Can work across multiple files coherently -  **Up-to-date:** Trained on recent Django/Python versions - 
Limitation: May be more verbose (explains a lot)

Why it's best for this project: - Can see your entire `models.py`, `views.py`, `settings.py` at once - Understands Django ORM deeply (queries, relationships, migrations) - Great at explaining *why* something works, not just *how* - Excellent for learning because it teaches as it codes





2. GPT-4 / GPT-4 Turbo (OpenAI) -  **Best for:** Quick code snippets, general programming questions -  **Context window:** 128K tokens (good for most files) -  **Code quality:** Very good, widely used -  **Speed:** Fast responses -  **Integrations:** Works with many tools (Cursor, GitHub Copilot Chat) -  **Django specifics:** Sometimes suggests outdated patterns -  **Explanation depth:** More concise (less teaching)





When to use: - Quick "how do I do X in Django?" questions - When you need a fast answer - When working in an IDE with GPT-4 integration






3. Gemini 1.5 Pro (Google) -  **Context window:** 1-2 million tokens (can see massive codebases) -  **Best for:** Analyzing large projects, documentation -  **Code quality:** Good but sometimes generic -  **Django expertise:** Less specialized in Django compared to Claude/GPT-4




When to use: - You need to analyze an entire large codebase at once - Reading and understanding documentation - Summarizing lots of code

Models That Fall Short for Django

GPT-3.5 -  Outdated Django patterns -  Smaller context window (can't see enough code) -  Makes more mistakes with modern Django features -  Less helpful explanations - **Verdict:** Skip it. The newer models are worth the cost/effort.

GitHub Copilot (base) -  Great for autocomplete while typing -  Not conversational (can't ask questions) -  Limited context (only sees current file) -  Can't explain why or debug - **Verdict:** Use alongside a chat model, not as replacement

Local Models (Llama, Mistral, CodeLlama) -  Privacy (runs on your computer) -  Free -  Smaller context windows -  Less accurate for Django-specific questions -  Slower on most hardware - **Verdict:** Great for privacy-sensitive work, but harder for beginners

Older Models (ChatGPT 3.5, Bard) -  Outdated training data -  Suggest deprecated Django patterns -  More errors, more time debugging - **Verdict:** Avoid. Using old models is like learning from a 2018 Django tutorial in 2025.

Practical Recommendations

For this insurance tracker project:

Best setup: 1. **Primary:** Claude Sonnet 3.5 or GPT-4 for development questions 2. **Secondary:** GitHub Copilot in VSCode for autocomplete 3. **Backup:** Keep both Claude and GPT-4 available - sometimes one explains better than the other

For different tasks:

Task	Best Model	Why
Learning Django	Claude Sonnet	Explains concepts deeply
Quick syntax	GPT-4	Fast, accurate
Debugging errors	Claude Sonnet	Better at reasoning through complex issues
Code review	Claude Opus	Most thorough analysis
Writing tests	Either Claude or GPT-4	Both excellent
Database queries	Claude Sonnet	Deep ORM understanding
CSS/HTML	Either works well	Less specialized knowledge needed
Deployment	Claude Sonnet	Better at DevOps reasoning

How to Access These Models

Claude: - Web: <https://claude.ai> (free tier available) - API: <https://console.anthropic.com> - IDE:

Use with Cursor or Claude Code

GPT-4: - Web: <https://chat.openai.com> (ChatGPT Plus - \$20/month) - API: <https://platform.openai.com> - IDE: GitHub Copilot Chat, Cursor

Cost Considerations: - **Free tier:** Claude has a generous free tier, GPT-4 via ChatGPT Plus - **API pay-as-you-go:** Usually \$0.50-\$5 per month for hobby projects - **Worth it?** Absolutely. The time saved pays for itself in the first hour.

The Bottom Line

For absolute beginners learning Django: → Use Claude Sonnet 3.5 as your primary assistant

Why? 1. It can see your entire project at once (200K context) 2. It explains *why*, not just *what* (better for learning) 3. Understands Django deeply (fewer outdated suggestions) 4. Patient with beginners (good at explaining concepts) 5. Great at catching mistakes before they become bugs


For experienced developers: → Use GPT-4 for speed, Claude for complex problems

Avoid: - GPT-3.5 (outdated) - Free/older models for learning (you'll learn wrong patterns) - Any model that can't see at least 50K tokens (too little context)

Pro tip: Ask the same complex question to both Claude and GPT-4, compare answers. You'll quickly learn which one you prefer for different tasks.

How to Ask Effective Questions

 **Bad request:** "Make my site better"

 **Good request:** "I want to add a feature where users can upload a PDF of their EOB document. The PDF should be stored with the EOEntry model and displayed as a download link in the policy detail page. Can you help me implement this?"

Why it's better: - Specific feature described - Clear acceptance criteria (upload PDF, store it, show download link) - Mentioned relevant model (EOEntry)

Example Feature Requests for Your Insurance Tracker

Feature 1: PDF Upload and Preview

Request to AI:

I need to add PDF upload functionality to my EOB entries. Here's what I need:

1. In the admin interface, add a file upload field for EOBEntry
2. Store PDFs in a 'eob_documents/' folder
3. On the policy detail page, show a "View EOB" link for each entry that has a PDF
4. When clicked, open the PDF in a new tab

My EOBEntry model already has an `eob_document` FileField. I just need help with:

- Configuring media file serving in development
- Adding the download link to the template

Feature 2: Monthly Spending Chart**Request to AI:**

I want to add a simple bar chart showing spending by month for the current year.

Requirements:

- Show on the policy detail page
- Use Chart.js (or suggest a simple alternative)
- X-axis: Months (Jan-Dec)
- Y-axis: Amount applied to deductible
- Use data from EOBEntry.date_of_service and EOBEntry.applied_to_deductible

Can you provide:

1. The Django view code to aggregate monthly spending
2. The template code with Chart.js
3. CDN link for Chart.js

Feature 3: Email Notifications**Request to AI:**

I want to send an email notification when the user reaches 80% of their deductible.

Requirements:

- Check in the EOBEntry save method
- Only send once per policy per year
- Use Django's built-in email functionality
- Email should include: policy name, total spent, remaining amount

Can you show me:

1. How to override the save() method on EOBEntry
2. How to check if we've already sent the notification (maybe add a field to InsurancePolicy?)
3. Sample email configuration for Gmail SMTP

Working Through Errors with AI

When you get an error, provide:

1. **The full error message** (copy from terminal)
2. **What you were trying to do**
3. **The relevant code** (the file and function where error occurred)

Example:

I'm getting this error when trying to view the policy detail page:

TemplateDoesNotExist at /policy/1/ insurance/policy_detail.html

I created the file at templates/insurance/policy_detail.html but Django can't find it.

Here's my settings.py TEMPLATES configuration:
[paste your TEMPLATES setting]

What am I missing?

Iterating on Features

Start small, then enhance:

1. **Version 1:** Simple table of EOB entries

2. **Version 2:** Add filtering by category
3. **Version 3:** Add date range filtering
4. **Version 4:** Add export to CSV
5. **Version 5:** Add visual charts

Ask AI to help with one version at a time. Test each version before moving to the next.

Learning from AI-Generated Code

Don't just copy-paste. Understand it:

When AI gives you code, ask follow-up questions: - "What does this line do?" - "Why did you use a ForeignKey instead of a CharField here?" - "What's the difference between `filter()` and `get()`?" - "Can you explain this list comprehension in simpler terms?"

Example conversation:

You: [Paste AI-generated code]

You: "Can you explain what `aggregate(total=Sum('applied_to_deductible'))` does line by line?"

AI: "Sure! This is a Django ORM method that performs a database aggregation: - `aggregate()` - Tells Django to calculate a summary value across multiple rows - `total=` - Names the result 'total' so you can access it as `result['total']` - `Sum('applied_to_deductible')` - Adds up all values in the 'applied_to_deductible' column - This generates SQL like: `SELECT SUM(applied_to_deductible) AS total FROM ...`"

Example AI Conversation for Your Next Feature

Let's say you want to add a "Quick Add EOB" form on the dashboard:

You: "I want to add a simple form on the dashboard page that lets me quickly add an EOB entry without going to the admin. The form should have: - Date of service (date picker) - Provider name (text input) - Category (dropdown) - Amount applied to deductible (number input) - Submit button

When submitted, it should create a new EOEntry and refresh the page. Can you help me implement this with Django forms?"

AI provides: Form class, view modification, template HTML

You test it

You: "It works! But I want to add client-side validation so the deductible amount can't be negative. How do I add that?"

AI provides: HTML5 validation attributes or JavaScript validation

You: "Perfect! One more thing - can the form automatically set the policy to the user's most recent policy instead of showing a dropdown?"

AI provides: View modification to auto-set policy

Keep iterating until perfect!

Next Steps

Congratulations! You now have a working Django project with a real-world use case. Here's what to do next:

Immediate Next Steps (This Week)

1. Add more EOB entries through the admin

- Try different categories
- Upload PDF documents (configure media serving first)
- Experiment with different statuses

2. Customize the dashboard

- Change colors to match your preference
- Add your insurance company's name
- Adjust the progress bar styling

3. Learn basic Git

```
cd ~/Coding/InsuranceTracker
git init
git add .
git commit -m "Initial insurance tracker implementation"
```

Why: This creates a save point. If you break something, you can always revert.

Short-term Enhancements (Next Few Weeks)

4. **Add a "Quick Add" form** (use AI to help!)
 - Form on dashboard to add EOB entries quickly
 - No need to use admin for common tasks
5. **Add filtering and search**
 - Filter EOB entries by date range
 - Search by provider name
 - Filter by category
6. **Create a reports page**
 - Monthly spending trends
 - Year-over-year comparison
 - Category breakdown pie chart
7. **Add export functionality**
 - Export EOB entries to CSV
 - Generate PDF summary reports
 - Email monthly summaries

Learning Resources

Django Official Tutorial: <https://docs.djangoproject.com/en/5.0/intro/tutorial01/> - More in-depth than this guide - Covers testing, forms, generic views - Work through it to solidify concepts

Python Basics: <https://www.learnpython.org/> - Free interactive Python tutorial - Learn loops, functions, classes - Will help you understand Django code better

Git Tutorial: <https://try.github.io/> - Interactive Git tutorial - Learn branching, merging, collaboration - Essential for any developer

HTML/CSS: <https://www.internetishard.com/> - Beautiful, beginner-friendly web tutorial - Learn to style your pages - Understand what Django templates generate

Common Pitfalls to Avoid

1. **Not using version control**
 - Always commit working code before making changes

- Use descriptive commit messages
- Commit often

2. Changing too many things at once

- Make one small change
- Test it
- Commit it
- Repeat

3. Ignoring errors

- Read error messages carefully
- They tell you exactly what's wrong and where
- Google the error message if you don't understand

4. Not asking for help

- Stack Overflow is your friend
- Django Discord/Slack communities
- AI assistants for quick questions
- Don't spend hours stuck - ask!

5. Skipping the documentation

- Django docs are excellent
- Reading docs is faster than trial-and-error
- Bookmark commonly used pages

Your Personal Development Path

Month 1: Get comfortable with Django basics - Follow official tutorial - Enhance your insurance tracker - Learn Git fundamentals - Deploy to a free hosting service (PythonAnywhere, Railway)

Month 2: Learn advanced Django - Django forms and validation - Custom user models - Django REST Framework for API - Background tasks with Celery

Month 3: Frontend and design - JavaScript basics - React or Vue.js (optional, for interactive UIs) - CSS frameworks (Tailwind CSS) - Responsive design

Month 4: Testing and deployment - Write unit tests - Integration testing - CI/CD with GitHub Actions - Production deployment (DigitalOcean, AWS)

Month 5-6: Build something new - Apply everything you learned - Build a different project from

scratch - Contribute to open source Django projects

Questions to Ask Yourself

As you build, periodically reflect:

1. **Do I understand why this code works?**
 - If not, ask AI or look it up
 - Don't move forward with "magic code"
2. **Is there a simpler way to do this?**
 - Django often has built-in solutions
 - Check the docs before reinventing
3. **What happens if this fails?**
 - Add error handling
 - Validate user input
 - Provide helpful error messages
4. **Could someone else understand this?**
 - Add comments for complex logic
 - Use descriptive variable names
 - Write docstrings for functions

Final Encouragement

You've just completed what many people never start. You: - Installed a complete development environment - Learned to use the terminal - Created a real database-backed web application - Learned Django fundamentals - Built something useful

This is a huge accomplishment!

Programming is a journey of continuous learning. Every expert was once a beginner who didn't give up. You'll encounter bugs, confusion, and frustration - that's normal. Every error message is a learning opportunity.

Remember: - Google is your friend (most errors have been solved before) - AI assistants are great tutors (ask them anything) - Communities are helpful (Django Discord, Reddit r/django) - Documentation is comprehensive (Django docs are excellent) - Practice makes perfect (build things, break things, fix things)

Getting Help

When you're stuck:

1. **Read the error message carefully**
2. **Google the exact error message**
3. **Check Django documentation**
4. **Ask AI assistant with full context**
5. **Post on Stack Overflow with code and error**
6. **Join Django Discord and ask there**

Stay Updated

Follow these resources: - Django Blog: <https://www.djangoproject.com/weblog/> - Real Python: <https://realpython.com/> (excellent tutorials) - Django Subreddit: [r/django](https://www.reddit.com/r/django/) - Django Discord: <https://discord.gg/xcRH6mN4fa>



Appendix: Command Reference

Terminal Commands

```

# Navigation
pwd                # Print working directory
ls                 # List files
ls -la             # List files with details
cd folder_name     # Change directory
cd ..              # Go up one level
cd ~               # Go to home directory

# File operations
mkdir folder_name  # Create directory
touch file.txt     # Create empty file
cp source.txt dest.txt # Copy file
mv old.txt new.txt # Rename/move file
rm file.txt        # Delete file
rm -r folder       # Delete folder

# View files
cat file.txt       # Show entire file
head file.txt      # Show first 10 lines
tail file.txt      # Show last 10 lines
less file.txt      # Paginated view (q to quit)

```

Git Commands

```

# Setup
git init           # Initialize repository
git config --global user.name "Your Name"
git config --global user.email "you@example.com"

# Basic workflow
git status         # See what changed
git add .          # Stage all changes
git add file.txt   # Stage specific file
git commit -m "Message" # Commit staged changes
git log            # View commit history
git log --oneline  # Compact log view

# Undo
git checkout file.txt # Discard changes to file
git reset HEAD file.txt # Unstage file
git revert commit_hash # Undo a commit safely

# Branches
git branch          # List branches
git branch feature_name # Create branch
git checkout feature_name # Switch to branch
git checkout -b feature # Create and switch
git merge feature_name # Merge branch

```

Python/Django Commands

```
# Python environment
python --version          # Check Python version
pip --version             # Check pip version
pip install package_name  # Install package
pip install -r requirements.txt # Install from file
pip freeze > requirements.txt # Save installed packages

# Pyenv
pyenv versions            # List installed Python versions
pyenv install 3.13.3      # Install Python version
pyenv global 3.13.3       # Set global Python version
pyenv virtualenv 3.13.3 project_name # Create virtual env
pyenv local project_name  # Activate virtual env for folder

# Django
django-admin startproject name . # Create project
python manage.py startapp name   # Create app
python manage.py runserver       # Start dev server
python manage.py runserver 8080  # Start on different port
python manage.py makemigrations  # Create migrations
python manage.py migrate         # Apply migrations
python manage.py createsuperuser # Create admin user
python manage.py shell           # Django Python shell
python manage.py collectstatic   # Collect static files
```

Docker Commands (Optional - For Later)

Note: You don't need Docker for this beginner project since we're using SQLite. These commands are here for reference when you're ready to use PostgreSQL or deploy to production.

```

# Container management
docker ps                # List running containers
docker ps -a             # List all containers
docker-compose up        # Start services
docker-compose up -d     # Start in background
docker-compose down      # Stop and remove containers
docker-compose logs      # View logs
docker-compose logs -f   # Follow logs









# Image management
docker images            # List images
docker pull image_name  # Download image
docker rmi image_name   # Remove image

# Cleanup
docker system prune      # Remove unused data
docker volume prune     # Remove unused volumes

```

You Did It!

You now have a fully functional Django project and the knowledge to keep building. The insurance tracker is just the beginning - you can apply these same concepts to any idea you have.

What you learned: -  Development environment setup -  Terminal navigation -  Git version control -  Django models, views, templates -  Database design -  Admin customization -  HTML/CSS basics -  Working with AI assistants

Your next project could be: - Personal finance tracker - Recipe manager - Habit tracker - Book library - Workout log - Project management tool - Anything you can imagine!

The skills you learned today transfer to any web project. Keep building, keep learning, and most importantly - have fun!

When Should I Upgrade to PostgreSQL?

You might wonder: "When do I switch from SQLite to PostgreSQL?" Here's an honest answer.

SQLite is Sufficient For

- **Personal projects** (just you using it)
- **Small team apps** (2-5 people, not using it constantly)
- **Development and testing** (always use SQLite locally)
- **Moderate data** (up to 100,000 records is totally fine)
- **Learning** (focus on Django, not database administration)

Real talk: Your insurance tracker will likely never outgrow SQLite. Unless you're tracking insurance for an entire company, SQLite is perfect.

Upgrade to PostgreSQL When

You have these specific problems:

1. **Many concurrent users** (20+ people writing to database simultaneously)
 - SQLite locks the entire database during writes
 - PostgreSQL handles concurrent writes gracefully
2. **Large datasets** (millions of records)
 - SQLite slows down on complex queries over huge datasets
 - PostgreSQL is optimized for big data
3. **Advanced features needed**
 - Full-text search across multiple columns
 - Complex JSON queries
 - Stored procedures
 - Row-level locking
4. **Production deployment requirements**
 - Many hosting platforms prefer PostgreSQL
 - Better backup and replication tools
 - Industry standard for production Django apps

How to Upgrade (When You're Ready)

Switching is actually simple! Here's what changes:

1. **Install PostgreSQL** (one-time setup)

```
# Mac
brew install postgresql@16
brew services start postgresql@16

# Or use Docker for easier management
docker run -d \
  --name postgres \
  -e POSTGRES_DB=insurance_tracker \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -p 5432:5432 \
  postgres:16
```

2. Install the PostgreSQL adapter

```
pip install psycopg2-binary
```

3. Update settings.py (just 8 lines)

```
# Change this:
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}

# To this:
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'insurance_tracker',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

4. Run migrations again

```
python manage.py migrate
```

That's it! Your code doesn't change at all. Django handles the differences.

My Recommendation

Start with SQLite. Don't even think about PostgreSQL until: - You have a specific problem SQLite can't solve - You're deploying to production with real users - You're 6+ months into your project and actually experiencing performance issues

Why? - One less thing to install and manage - One less thing to break - More time focusing on building your actual app - When you do need PostgreSQL, switching takes 10 minutes

Instagram started with SQLite. Stack Overflow started with SQLite. You're in good company.

Focus on building, not infrastructure.

"Wait, Should I Use Flask Instead?"

You might hear people say "Flask is simpler for beginners." Let's have an honest conversation about this.

The "Flask is Simpler" Myth

People say Flask is simpler, and that's... **half true**:

Flask Hello World:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Hello World"
```

Django Hello World:

```
# Needs settings.py, urls.py, views.py, etc.
# More files, more structure
```

Flask **looks** simpler initially. But once you need: - Database models → Install SQLAlchemy, configure it - User login → Install Flask-Login, set up sessions - Forms → Install Flask-WTF, configure CSRF - File uploads → Configure upload folders, validation - Admin panel → Install

Flask-Admin, configure every model

Suddenly you're Googling "Flask best practices" and getting 10 different answers.

Django vs Flask for This Project

For the insurance EOB tracker specifically:

Feature	Django	Flask
Time to working app	30 minutes	3-4 hours
Lines of code	~200	~500+
Admin interface	Free, automatic	Build it yourself
User authentication	Built-in	Install + configure extension
Database migrations	Built-in	Install Alembic, configure
Form validation	Built-in	Install Flask-WTF
File uploads	Configured	Configure manually
Security (CSRF, XSS)	Enabled by default	Configure yourself

Django Advantages for This Project

- 1. Admin Interface (Huge Time Saver)** - Django: Free, automatic admin panel to manage data - Flask: You'd need to build every form, table, and CRUD interface manually
- 2. Less Code to Write** - Django EOB tracker: ~200 lines of Python - Flask equivalent: ~500+ lines (need to add Flask-SQLAlchemy, Flask-Login, Flask-WTF, etc.)
- 3. Fewer Decisions** - Django: "There's one obvious way" - just follow the structure - Flask: "Should I use SQLAlchemy or Peewee? WTFORMS or Flask-WTF? Jinja templates where?" = decision fatigue
- 4. Better for Learning Full-Stack Concepts** - Django teaches you how complete web apps work as a system - Flask teaches you Python fundamentals, but you'll need to learn how to connect the pieces

When Flask Is Actually Better








Flask shines for: - Simple APIs (just a few endpoints) - Learning Python basics first (less "magic") - Microservices - When you want maximum control - Very simple single-page apps - When you already know web development and want lightweight

Where Beginners Get Stuck

Django beginners struggle with: - "What's a migration?" (but you'll learn this quickly) - "Where does this file go?" (but the structure prevents chaos later) - "Why so much boilerplate?" (but it's solving problems you don't know you have yet)

Flask beginners struggle with: - "Which packages should I use?" (analysis paralysis) - "How do I structure this?" (everyone's app looks different) - "Why isn't this working?" (misconfigured extensions) - "How do I add user accounts?" (watch a 2-hour tutorial) - "Did I configure security correctly?" (easy to make mistakes)

The Uncomfortable Truth

For your insurance tracker, Flask would mean: -  More control over every detail -  Less "magic" happening behind the scenes -  Lighter weight (fewer dependencies) -  3-4x more code to write -  More security mistakes possible -  No admin interface (you'd build forms for everything) -  Longer time to working product

Is that worth it? Or do you want to track your medical expenses sometime this month?

The Real Question

What's your actual goal?

1. **"I want to understand how web apps work from scratch"** → Flask is better. You'll wire everything up manually and understand each piece.
2. **"I want to build this insurance tracker and actually use it"** → Django is better. Focus on your domain logic, not plumbing.
3. **"I want to become a professional developer"** → Learn both eventually. Start with Django for structure, then Flask shows you what Django does behind the scenes.

My Recommendation

Stick with Django for this project because:

1. You want to build something **useful quickly** (track real insurance data)
2. The admin interface lets you focus on logic, not building forms
3. You have a template already set up
4. Django's structure prevents beginner mistakes
5. When you understand Django, Flask will be trivial to learn later

Consider Flask later when: - You understand databases, routing, templates, authentication - You want to build a simple API - You want to learn "how it works under the hood" - You're building a microservice that does one thing

The Analogy

- **Django** = Learning to drive with an automatic transmission car that has GPS, parking assist, and lane-keeping
- **Flask** = Learning to drive with a manual transmission and a paper map

Both get you there, but for a total beginner trying to accomplish a real goal (not just learn), the automatic car gets you driving safely much faster.

The Bottom Line

The best framework is the one that gets you building.

Paralysis by analysis kills more beginner projects than choosing the "wrong" framework. Pick one, build something, ship it. You can always rebuild in Flask later if you want - but you'll have a working product in the meantime.

Django isn't "training wheels" - it's the framework that Instagram, Pinterest, and The Washington Post use in production. It scales from beginner projects to billion-user platforms.

Start with Django. Build your insurance tracker. Learn the concepts. Then evaluate if Flask makes sense for your next project.



Reference Sections

How to use these sections: These are detailed references you can copy/paste into AI chat

when you need help. For example, if you're getting an error, copy the "Troubleshooting" section + your error message and paste it to Claude or ChatGPT.

Troubleshooting & Common Errors

Understanding Django Error Pages

When something goes wrong, Django shows a detailed error page (when `DEBUG=True`). Here's how to read it:

Anatomy of a Django error:

```
Exception Type: ValueError
Exception Value: invalid literal for int() with base 10: 'abc'
Exception Location: /path/to/your/file.py, line 42

Request Method: GET
Request URL: http://127.0.0.1:8000/policy/abc/
```

What to look at: 1. **Exception Type** - What kind of error (ValueError, ImportError, etc.) 2. **Exception Value** - What went wrong in plain English 3. **Exception Location** - Which file and line number 4. **Request URL** - What page you were trying to access

The traceback (below the error) shows the path your code took to get to the error. Read from bottom to top - the bottom is where it broke.

Common Errors & Solutions

1. ModuleNotFoundError: No module named 'django'

Error:

```
ModuleNotFoundError: No module named 'django'
```

What it means: Python can't find Django. Your virtual environment isn't activated or Django isn't installed.

Solutions:

```
# Check if virtual environment is active
which python
# Should show path with your virtualenv name

# If not active, activate it
pyenv local insurance-tracker

# Verify Django is installed
pip list | grep Django

# If Django is missing, install it
pip install -r requirements.txt
```

2. Port Already in Use (8000)

Error:

```
Error: That port is already in use.
```

What it means: Something is already running on port 8000 (probably an old Django server you forgot to stop).

Solutions:

```
# Option 1: Find and kill the process
lsof -ti:8000 | xargs kill -9

# Option 2: Use a different port
python manage.py runserver 8001

# Option 3: Close your terminal and open a new one
# (kills all processes from that terminal)
```

3. ImproperlyConfigured: The SECRET_KEY setting must not be empty

Error:

```
django.core.exceptions.ImproperlyConfigured: The SECRET_KEY setting must not be empty.
```

What it means: Django can't find the SECRET_KEY in settings.py

Solutions:

```
# Open config/settings.py
# Find the SECRET_KEY line (around line 23)

# Make sure it looks like this:
SECRET_KEY = 'django-insecure-some-long-random-string-here'

# If it's trying to load from environment:
SECRET_KEY = os.getenv('SECRET_KEY', 'django-insecure-fallback-key-for-dev')
```

4. DoesNotExist: InsurancePolicy matching query does not exist

Error:

```
apps.insurance.models.InsurancePolicy.DoesNotExist: InsurancePolicy matching query does not exist.
```

What it means: You're trying to access a database record that doesn't exist.

Common causes: - Tried to access `/policy/1/` but there's no policy with ID=1 - Deleted data from admin but code still references it - Wrong ID in URL

Solutions:

```
# In views.py, use get_object_or_404 instead of .get()
from django.shortcuts import get_object_or_404

# Change this:
policy = InsurancePolicy.objects.get(id=policy_id)

# To this (shows 404 page instead of error):
policy = get_object_or_404(InsurancePolicy, id=policy_id)
```

5. TemplateDoesNotExist: insurance/dashboard.html

Error:

```
django.template.exceptions.TemplateDoesNotExist: insurance/dashboard.html
```

What it means: Django can't find your template file.

Common causes: - Template is in wrong folder - Typo in template name - TEMPLATES setting misconfigured

Solutions:

```
# Check your template location
ls templates/insurance/

# Make sure structure is:
# templates/
#   insurance/
#     dashboard.html
#     policy_detail.html

# If template exists but still not found, check settings.py:
TEMPLATES = [
    {
        'DIRS': [BASE_DIR / 'templates'], # Make sure this line exists
        ...
    }
]
```

6. No changes detected (when running makemigrations)

Error:

```
$ python manage.py makemigrations
No changes detected
```

What it means: Django doesn't see any changes to your models, even though you changed them.

Common causes: - App not in INSTALLED_APPS - Syntax error in models.py - Forgot to save the file

Solutions:

```
# 1. Check if your app is in INSTALLED_APPS (settings.py)
# Should have: 'apps.insurance'

# 2. Try specifying the app name
python manage.py makemigrations insurance

# 3. Check for syntax errors
python manage.py check

# 4. Make sure you saved models.py (CMD+S)
```

7. OperationalError: no such table: insurance_eobentry

Error:

```
django.db.utils.OperationalError: no such table: insurance_eobentry
```

What it means: You're trying to query a database table that doesn't exist yet.

Solutions:

```
# Run migrations to create tables
python manage.py migrate

# If that doesn't work, check if migrations exist
ls apps/insurance/migrations/

# If no migration files (except __init__.py), create them
python manage.py makemigrations insurance

# Then run migrations
python manage.py migrate
```

8. CSRF Verification Failed

Error:

Forbidden (403)
CSRF verification failed. Request aborted.

What it means: Django's security feature blocking your form submission.

Solutions:

```
<!-- In your form template, make sure you have: -->  
<form method="post">  
    {% csrf_token %} <!-- ADD THIS LINE -->  
    <!-- your form fields -->  
</form>
```

9. Static Files Not Loading (CSS/Images)

Problem: Page loads but no styling, images broken.

Solutions:


```
# 1. Check settings.py has these lines:
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / 'static']

# 2. Make sure folder structure is:
# static/
#   css/
#     style.css
#   js/
#   images/

# 3. In templates, load static files:
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>

# 4. Check urls.py includes this (for development):
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # your urls
]

if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL,
        document_root=settings.STATICFILES_DIRS[0])
```

10. Virtual Environment Not Activating

Problem: Commands show wrong Python version or "django-admin not found"

Solutions:

```
# Check what python you're using
which python
python --version

# Should show your virtualenv path and Python 3.13.3

# If wrong, reactivate:
cd ~/Coding/InsuranceTracker
pyenv local insurance-tracker

# If that doesn't work, check .python-version file exists
ls -la | grep python-version

# If missing, recreate it
echo "insurance-tracker" > .python-version

# Close and reopen terminal, then cd back into project
```

How to Debug Like a Pro

Step-by-step debugging process:

1. **Read the full error message** (don't just look at the first line)
2. **Find the line number** where error occurred
3. **Open that file** and look at that line
4. **Look at what variables exist** at that point
5. **Add print statements** to see values
6. **Google the error** (include "django" in search)
7. **Ask AI for help** (provide full error + relevant code)

Using print() to debug:

```
def policy_detail(request, policy_id):
    print(f"Policy ID requested: {policy_id}") # Debug
    print(f"Type: {type(policy_id)}") # Debug

    policy = get_object_or_404(InsurancePolicy, id=policy_id)
    print(f"Found policy: {policy}") # Debug

    # If you reach here, you'll see prints in terminal
```

Using Django shell for testing:

```
python manage.py shell

>>> from apps.insurance.models import InsurancePolicy
>>> policies = InsurancePolicy.objects.all()
>>> print(policies)
>>> # Test queries here without needing the web server
```

Basic Git Workflow

Your First Commits

After your insurance tracker works, save your progress with Git:

```
# 1. See what changed
git status

# You'll see:
# - Red files = not staged
# - Green files = staged (ready to commit)

# 2. Stage all changes
git add .

# Or stage specific files
git add apps/insurance/models.py
git add templates/insurance/dashboard.html

# 3. Commit with a descriptive message
git commit -m "Add insurance tracker with EOB models and dashboard"

# 4. View your commit history
git log --oneline
```

Daily Git Workflow

After each working feature:

```
# 1. Save your work
git add .
git commit -m "Add category spending chart to policy detail page"

# 2. Before starting something new, check status
git status

# Should show: "working tree clean"
```

Common Git Tasks

See what changed:

```
# Show which files changed
git status

# Show actual code differences
git diff

# Show differences for one file
git diff apps/insurance/views.py
```

Undo mistakes:

```
# Undo changes to ONE file (before committing)
git checkout -- apps/insurance/models.py

# Undo ALL changes (before committing) - CAREFUL!
git checkout -- .

# Unstage a file (keep changes, just remove from staging)
git reset HEAD apps/insurance/models.py

# Go back to previous commit (creates new commit)
git revert HEAD

# See last 5 commits
git log --oneline -5

# Go back to a specific commit (DESTRUCTIVE - lose changes)
git reset --hard abc123
```

Branches (for experimenting):

```
# Create a branch for new feature
git branch add-export-feature

# Switch to that branch
git checkout add-export-feature

# Or do both at once
git checkout -b add-export-feature

# Make changes, commit them
git add .
git commit -m "Add CSV export functionality"

# Switch back to main
git checkout main

# Merge the feature in
git merge add-export-feature

# Delete the feature branch (after merging)
git branch -d add-export-feature
```

When Things Go Wrong

"I committed something by mistake!"

```
# Undo last commit, keep the changes
git reset --soft HEAD~1

# Now you can edit files and commit again
```

"I broke everything, go back!"

```
# See history
git log --oneline

# Find the good commit (like abc123), then
git reset --hard abc123

# WARNING: This deletes all changes after that commit!
```

"I want to save work but try something else"

```
# Stash your changes (temporary storage)
git stash

# Do other work, commit it...

# Bring back your stashed changes
git stash pop
```

Environment Variables & Secrets

What Are Secrets?

Secrets are sensitive values you don't want in Git: - `SECRET_KEY` (Django's encryption key) - Database passwords - API keys (Stripe, SendGrid, etc.) - Email passwords

Why not commit them? - Your Git repo might become public - Hackers scan GitHub for leaked keys - Different environments need different values (dev vs production)

Using .env Files

1. Create a `.env` file in your project root:

```
# .env
SECRET_KEY=django-insecure-your-secret-key-here
DEBUG=True
DATABASE_URL=sqlite:///db.sqlite3
EMAIL_HOST_PASSWORD=your-email-password
```

2. Add `.env` to `.gitignore`:

```
# .gitignore
.env
*.pyc
__pycache__
db.sqlite3
```

3. Load .env in settings.py:

```
# At the top of config/settings.py
from pathlib import Path
import os
from dotenv import load_dotenv

load_dotenv() # Load .env file

# Now use environment variables
SECRET_KEY = os.getenv('SECRET_KEY', 'fallback-key-for-dev')
DEBUG = os.getenv('DEBUG', 'False') == 'True'

# For database
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

4. Create `.env.example` to commit:

```
# .env.example (commit this)
SECRET_KEY=your-secret-key-here
DEBUG=True
DATABASE_URL=sqlite:///db.sqlite3
EMAIL_HOST_PASSWORD=your-password
```

This shows others what variables they need without exposing real values.

Deployment Basics

Free Hosting Options

Best free platforms for beginners:

1. **PythonAnywhere** - Easiest for Django
2. **Railway** - Modern, Git-based deployment
3. **Render** - Free tier, PostgreSQL included
4. **Fly.io** - Fast, Docker-based

Quick Deploy to PythonAnywhere

Step-by-step:

1. **Sign up:** <https://www.pythonanywhere.com> (free tier)
2. **Open a Bash console** (on dashboard)
3. **Clone your project:**

```
git clone https://github.com/yourusername/InsuranceTracker.git
cd InsuranceTracker
```

4. **Create virtualenv:**

```
mkvirtualenv --python=/usr/bin/python3.10 insurance-tracker
pip install -r requirements.txt
```

5. **Run migrations:**

```
python manage.py migrate
python manage.py createsuperuser
```

6. **Configure web app:**

- Go to "Web" tab
- Click "Add a new web app"
- Choose "Manual configuration"
- Python 3.10
- Set source code: `/home/yourusername/InsuranceTracker`
- Set virtualenv: `/home/yourusername/.virtualenvs/insurance-tracker`

7. **Edit WSGI file** (link on Web tab):


```
import os
import sys

path = '/home/yourusername/InsuranceTracker'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'config.settings'

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

8. Update settings.py for production:

```
ALLOWED_HOSTS = ['yourusername.pythonanywhere.com']
DEBUG = False
```

9. Reload web app (green button on Web tab)

10. Visit: <https://yourusername.pythonanywhere.com>

What Changes for Production

Development vs Production:

Setting	Development	Production
DEBUG	True	False
ALLOWED_HOSTS	[]	['yourdomain.com']
Database	SQLite	PostgreSQL
SECRET_KEY	Hardcoded	From environment
Static files	Dev server serves them	Collected with collectstatic

Key changes needed:

```
# settings.py for production
DEBUG = False
ALLOWED_HOSTS = ['yourdomain.com', 'www.yourdomain.com']
SECRET_KEY = os.getenv('SECRET_KEY') # From environment

# Static files for production
STATIC_ROOT = BASE_DIR / 'staticfiles'

# Run this before deploying:
# python manage.py collectstatic
```

What to Build Next

Next Features for Insurance Tracker

Easy (do these first): 1. **Search EOB entries** by provider name or date 2. **Filter by status** (pending, paid, disputed) 3. **Sort table** by date, amount, or category 4. **Add notes** to EOB entries 5. **Mark as paid** button (update status quickly)

Intermediate: 6. **Export to CSV** (download all EOBs as spreadsheet) 7. **Date range filter** (show EOBs from Jan-March) 8. **Monthly spending chart** (bar chart with Chart.js) 9. **Category pie chart** (visual breakdown) 10. **Multiple policies** (track family members separately)

Advanced: 11. **PDF upload** (attach EOB documents) 12. **OCR text extraction** (read amounts from PDF automatically) 13. **Email notifications** (alert when deductible met) 14. **Recurring expenses** (auto-create monthly entries) 15. **Year-over-year comparison** (compare 2024 vs 2025)

Learning Path After This Guide

Week 1-2: Django Fundamentals - Official Django Tutorial (Parts 1-7): <https://docs.djangoproject.com/en/stable/intro/tutorial01/> - Django Girls Tutorial: <https://tutorial.djangogirls.org/> - Practice: Add more features to insurance tracker

Week 3-4: Advanced Django - Django Forms (not just admin) - Class-Based Views - Django REST Framework (APIs) - User authentication customization

Month 2: Frontend Skills - JavaScript basics - HTMX (dynamic pages without heavy

JavaScript) - Tailwind CSS or Bootstrap (styling frameworks) - Chart.js or D3.js (data visualization)

Month 3: Professional Skills - Writing tests (pytest, unittest) - Continuous Integration (GitHub Actions) - Docker basics - PostgreSQL deep dive

Month 4+: Build Your Own Projects - Recipe manager - Habit tracker - Personal finance dashboard - Project management tool - Whatever solves YOUR problems

Free Learning Resources

Best Django Resources: - **Django Docs:** <https://docs.djangoproject.com/> (always your first stop) - **Django Girls Tutorial:** <https://tutorial.djangogirls.org/> (beginner-friendly) - **Real Python:** <https://realpython.com/> (tutorials, courses) - **Simple is Better Than Complex:** <https://simpleisbetterthancomplex.com/> (great blog) - **Corey Schafer YouTube:** Django tutorial series (excellent quality)

Python Fundamentals: - **Python.org Tutorial:** <https://docs.python.org/3/tutorial/> - **Automate the Boring Stuff:** <https://automatetheboringstuff.com/> (free book) - **CS50 Python:** <https://cs50.harvard.edu/python/> (Harvard's free course)

Frontend: - **MDN Web Docs:** <https://developer.mozilla.org/> (HTML, CSS, JavaScript) - **JavaScript.info:** <https://javascript.info/> (modern JavaScript) - **CSS Tricks:** <https://css-tricks.com/> (CSS solutions)

Git: - **Pro Git Book:** <https://git-scm.com/book/en/v2> (free, comprehensive) - **Learn Git Branching:** <https://learngitbranching.js.org/> (interactive)

Communities: - **r/django** (Reddit) - **Django Discord:** <https://discord.gg/xcRH6mN4fa> - **Stack Overflow:** Tag your questions with `django` - **Django Forum:** <https://forum.djangoproject.com/>

Avoiding Tutorial Hell

Tutorial Hell = doing tutorial after tutorial without building your own projects.

Signs you're stuck: - Completed 5+ tutorials but can't build anything alone - Always need to follow along, can't code from scratch - Afraid to start your own project

How to escape: 1. **Stop doing tutorials** (seriously) 2. **Pick a project idea** that solves YOUR problem 3. **Build it** (you'll get stuck - that's good!) 4. **Google when stuck** (specific questions)

get specific answers) 5. **Ask AI for help** (with context from your actual project) 6. **Finish it** (even if it's ugly - shipping matters) 7. **Repeat** with a harder project

The rule: After this guide, build 3 projects on your own before doing another tutorial.

? How to Ask for Help Effectively

Creating Minimal Reproducible Examples

Bad question: "My Django app doesn't work. Help?"

Good question: "Getting DoesNotExist error when accessing policy detail page. Here's my view (code), model (code), and traceback (error). Policy with ID=1 exists in admin."

Stack Overflow Etiquette

Before posting: 1. Google the error message 2. Search Stack Overflow for similar questions 3. Try debugging yourself (see Troubleshooting section)

When posting: 1. **Title:** Specific error + technology (e.g., "Django DoesNotExist error when querying InsurancePolicy") 2. **Problem:** What you're trying to do 3. **Expected:** What should happen 4. **Actual:** What actually happens (include error) 5. **Code:** Minimal code that reproduces issue 6. **Tried:** What you've already attempted

Template:

```
## Problem
I'm trying to display a policy detail page, but getting a DoesNotExist error.

## Expected Behavior
Should show policy details for policy ID 1

## Actual Behavior
```

DoesNotExist: InsurancePolicy matching query does not exist.

```
## Code
```python
views.py
def policy_detail(request, policy_id):
 policy = InsurancePolicy.objects.get(id=policy_id)
 return render(request, 'insurance/policy_detail.html', {'policy':
policy})
```

## What I've Tried

- Verified policy with ID=1 exists in Django admin
- Checked INSTALLED\_APPS includes 'apps.insurance'
- Ran migrations successfully

## Environment

- Django 5.0.1
- Python 3.13.3
- SQLite database

---

### Asking AI for Help

\*\*Give context:\*\*

I'm building a Django insurance tracker following this guide: [paste relevant section]

I'm trying to [what you want to do]

Here's my current code: [paste models.py, views.py, and template]

I'm getting this error: [paste full error]

What's wrong and how do I fix it? ```

**Follow-up questions:** - "Can you explain why that works?" - "Is there a better way to do this?"  
- "What are the security implications?"