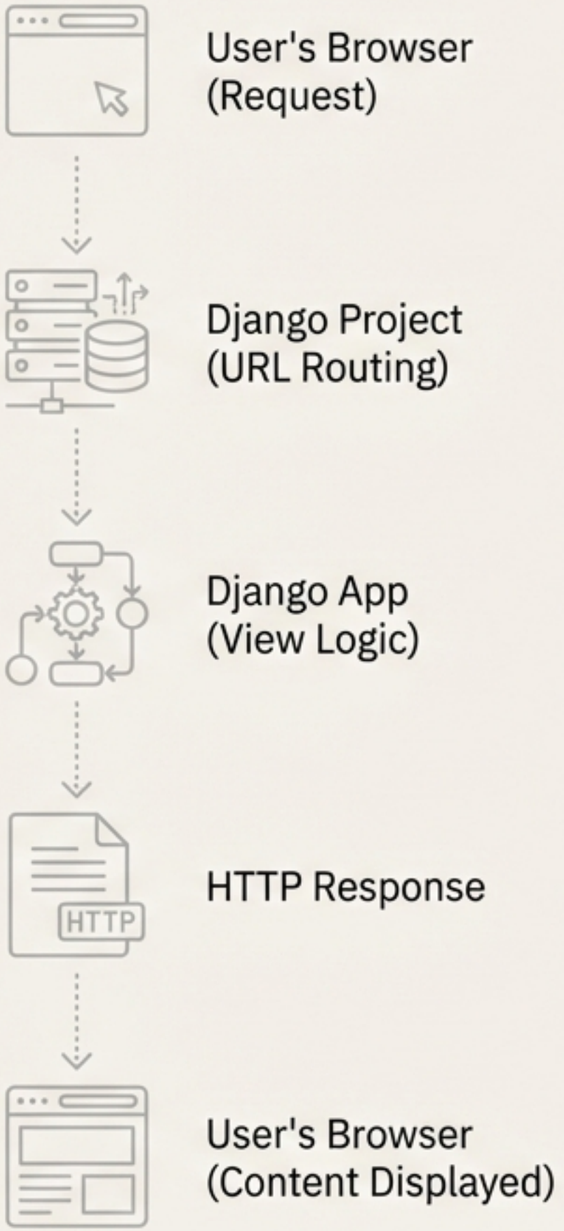
From Command Line to “Hello, World!”

A step-by-step guide to your first Django web application.

Our Goal: The Request-Response Cycle

Before we write any code, let's understand the journey of a web request in Django. Our mission is to build the components that handle this cycle, turning a user's browser request into a 'Hello, World!' response.

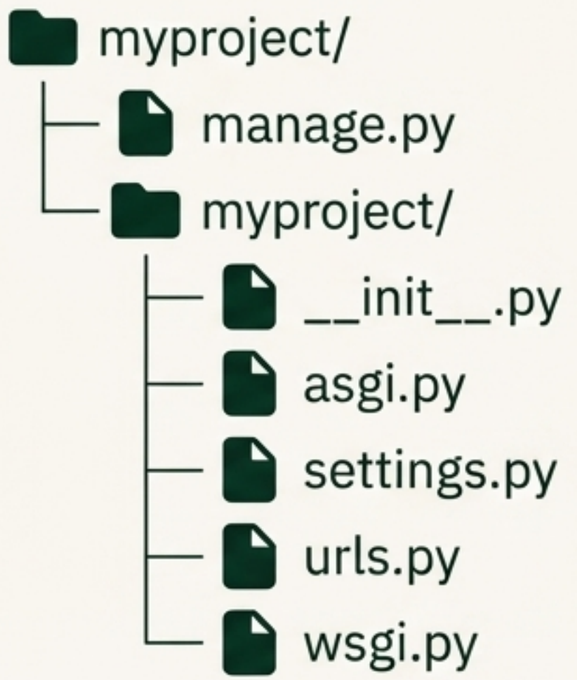
We will build each part of this chain, step by step.



Step 1: Laying the Foundation with a Project

A Django project is the main container for your entire web application. It holds the configuration, URL maps, and all the functional applications. We create it from the terminal.

```
JetBrains Mono
$ django-admin startproject myproject
```



Understanding Your Project's Blueprint

Django generates several files, but three are critical for getting started:

`manage.py`

The Task Manager

Your command-line utility for interacting with the project. You'll use this to run the server, create apps, and manage the database.

`settings.py`

The Control Panel

The central configuration file. It holds database settings, installed apps, middleware, and more.

`urls.py`

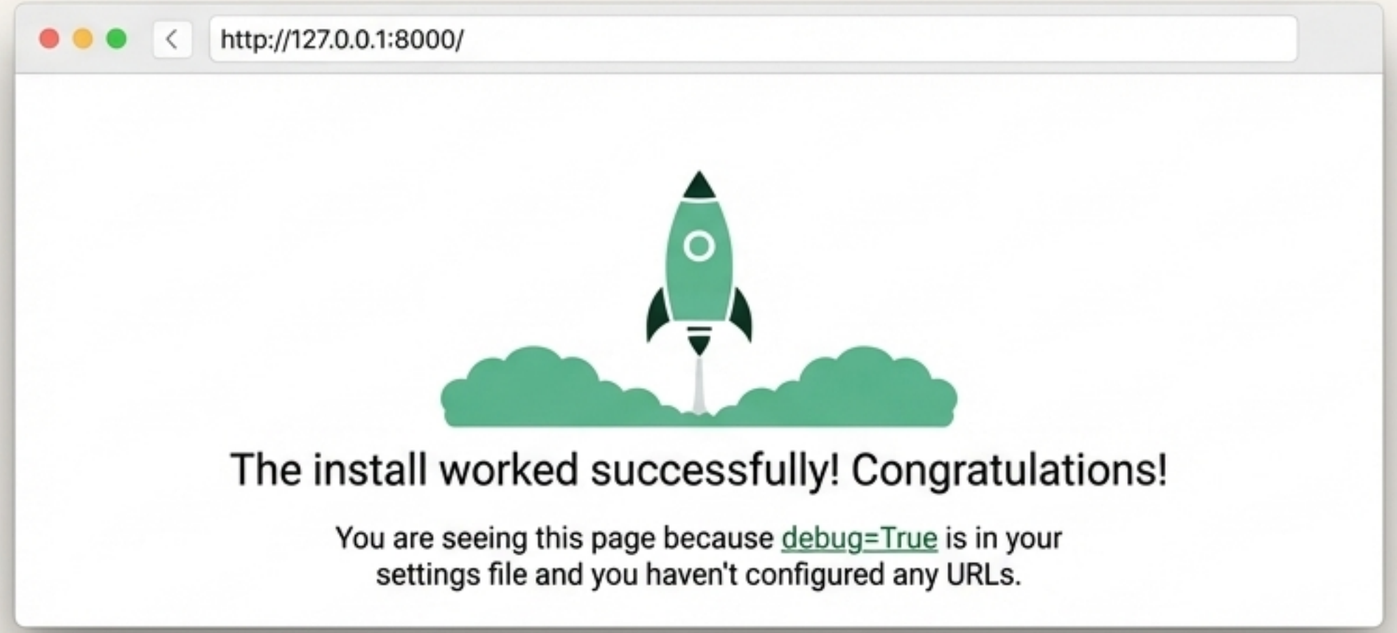
The Main Switchboard

The entry point for all URL requests. It directs incoming traffic to the correct part of your application.

Activating the Development Server

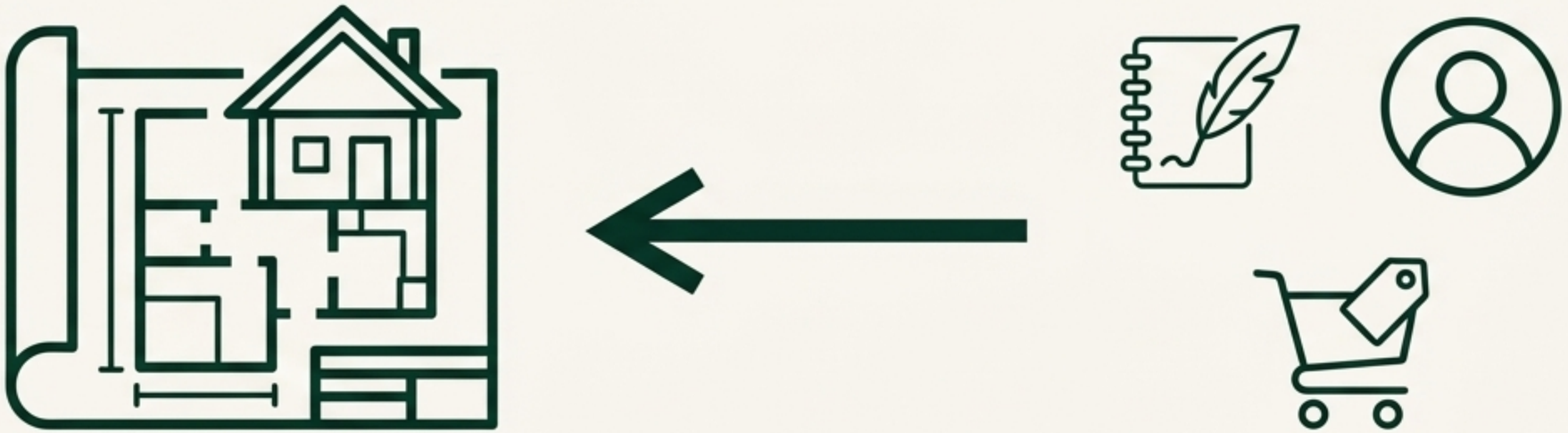
Django includes a lightweight web server for development. Let's start it to see our new, empty project live in the browser.

```
$ cd myproject
$ python manage.py runserver
```



The Core Architecture: Projects vs. Apps

A Django project is a single website. An app is a self-contained feature within that website. The goal is to create small, reusable apps that do one thing well.



Project

The entire website's configuration and container

Apps

Reusable features like a blog, user accounts, or a shop.

Step 2: Building Our First Functional App

Now we'll create an app to house the logic for our 'Hello, World!" page. We use manage.py again for this task.

```
$ python manage.py startapp pages
```

```
myproject/
├── manage.py
├── myproject/
│   └── ...
└── pages/
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations/
    ├── models.py
    ├── tests.py
    └── views.py
```

Registering the App with the Project

Creating the app's files isn't enough. We must explicitly tell the main project that this new `pages` app exists and should be included. We do this in the `settings.py` file.

myproject/settings.py

```
# myproject/settings.py

INSTALLED_APPS = [
      'django.contrib.admin',
      'django.contrib.auth',
      'django.contrib.contenttypes',
      'django.contrib.sessions',
      'django.contrib.messages',
      'django.contrib.staticfiles',
      'pages',                                  # Our new app
]
```

Step 3: Crafting the Response in a View

A view is a Python function that handles a web request and returns a response. All our application logic lives here. We will write our first view in the `pages/views.py` file.

pages/views.py

```python
# pages/views.py

from django.http import HttpResponse

def home_page_view(request):
    return HttpResponse("Hello, World!")
```

Deconstructing the View Function

Let’s break down the three key parts of our simple view.

```
from django.http import HttpResponse

def home_page_view(request):
    return HttpResponse("Hello, World!")
```

We import the `HttpResponse` class, which lets us send a simple text response back to the browser.

We define a function. Django automatically passes it a `request` object containing information about the user's request.

We create an instance of `HttpResponse` with our desired message and return it. This completes the cycle.

The Missing Link: Connecting a URL to Our View

We've created a view function, but it's currently unreachable. How does Django know to execute our `home_page_view` function when a user visits a specific URL?

The answer is Django's URLconf system.

Step 4: Mapping URLs to Views

The best practice is a two-step process. The main project's `urls.py` acts as a traffic controller, directing requests for a certain path (e.g., `/`) to a dedicated `urls.py` file inside our `pages` app.

Creating the App's URL Map

First, we create a new `urls.py` file inside our `pages` app. This file will contain all URL patterns specific to this app.

pages/urls.py

```
# pages/urls.py

from django.urls import path
from .views import home_page_view

urlpatterns = [
    path('', home_page_view, name='home'),
]
```

The `path('', ...)` means this view will handle the root URL of the app (e.g., `/` if the project delegates that path to us).

Wiring the App into the Main Project

Finally, we edit the main `myproject/urls.py` file. We use the `include` function to tell the project to hand off any requests for a specific path to our app's `urls.py` file.

myproject/urls.py

```
# myproject/urls.py

from django.contrib import admin
from django.urls import path, include  # 1

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')), # 2
]
```

1. Import the `include` function.

2. This line tells Django: for any request to the root URL (''), hand it over to the `urls.py` file inside the `pages` app for processing.

The Journey Complete: From Request to ‘Hello, World!’

We have successfully built every component needed to handle a web request. The user’s request now flows through our project, is routed to our app, processed by our view, and returned as a response.