</> FOR DEVELOPERS

# Build & Ship a Production SaaS with Django

A practical, end-to-end guide to building, deploying, and scaling a multi-tenant SaaS application with Django.


**BUILT WITH**

- django
- Python
- PostgreSQL
- Redis
- Docker
- Tailwind CSS

Multi-Tenant Architecture | Authentication & Authorization | Billing & Subscriptions | CI/CD & Deployment

TECHNOVIZE PUBLISHING

</> djangozen.com

# Build & Ship a Production SaaS with Django

*From an empty repo to a deployed, billable multi-tenant product*

**KC Ramo**

Technovize Publishing

technovize.com

First Edition, 2026

Published by Technovize Publishing · Amsterdam, The Netherlands · technovize.com

# Contents

# Preface

*Why this book exists*

Most Django material teaches you to build features. Very little teaches you to build a business — a product that many customers pay for every month, whose data must never leak between them, whose payments must never quietly fail, and which must stay up while you sleep. That gap is what this book exists to close. It is not a reference to be dipped into, nor a collection of disconnected recipes; it is a single, continuous build of a real Software-as-a-Service product, from an empty directory to a deployed, billable application, with every production concern addressed along the way.

## Who should read this book

This book is for developers who know Django and want to ship a SaaS. If you have built at least one small Django application — you are comfortable with models, views, templates, the ORM, and migrations — you have the foundation you need. You do not need prior experience with multi-tenancy, Stripe, Celery, Docker, or production operations; each is introduced where the build requires it and explained from first principles. If you have ever finished a tutorial and wondered how to turn it into something people pay for, this book is written for you.

## What you will build

Rather than teach concepts in the abstract, we build one complete product and let the concepts arise from real needs. That product is TaskFlow, a multi-tenant team task-management SaaS: organizations sign up, invite teammates, create projects, and track tasks, on a subscription. The domain is deliberately familiar, because the value is not in the novelty of task management but in the machinery every SaaS shares — tenant isolation, authentication and roles, subscription billing, background jobs, transactional email, security, deployment, and operations. Learn to build that machinery here and you can lift it, nearly unchanged, into a product of your own.

## How the book is organized

The book follows the order you would actually build in, grouped into five parts. Part I, Foundations, sets up the project, its configuration, and the service-layer architecture we use throughout. Part II, Users, Tenancy & the Product, adds authentication, turns organizations into isolated tenants, and builds the core product. Part III, Monetization & Background Work, wires up Stripe billing, Celery background jobs, and transactional email. Part IV, Access, Experience & Quality, covers authorization and roles, the frontend, and testing. Part V, Shipping & Operating, hardens, deploys, observes, scales, and launches the product. Each chapter ends with a summary and a set of key takeaways, and the appendices consolidate the checklists and references you will reach for again.

## Conventions used in this book

Code appears in monospaced blocks and is shown in the context where it lives, with imports and boilerplate trimmed once a pattern is established so the important lines stand out. Where a snippet omits detail for brevity, the surrounding text says what it stands for. Filenames are given relative to the project root, so `apps/organizations/services.py` means exactly that path. Commands are shown as you would type them in a terminal. Throughout, a recurring principle guides the choices: production-ready means being honest about trade-offs. Where a decision matters, the book explains the alternatives and why we chose what we chose, so that you can make a different call when your context differs.

## A note on the code

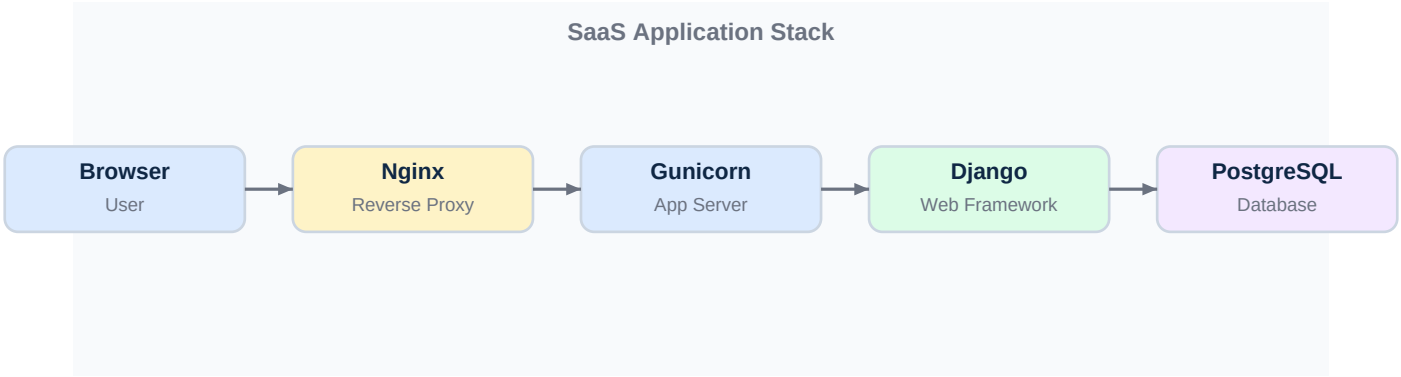
The code in this book is written to be understood and adapted, not merely copied. It is correct and idiomatic, but the goal is always to convey the reasoning behind it, so that you can adjust it to your own product rather than treat it as a black box. Django, Python, Stripe, and the wider ecosystem evolve; the specific APIs may shift over time, but the patterns and principles — the service layer, the tenant boundary, webhooks as the source of truth, testing the properties that matter — are durable. Learn those, and the details will follow. With that, let us begin where every real product begins: with a clear picture of what we are building and why it is shaped the way it is.

PART 1

# Foundations

CHAPTER 1

# Introduction




There is a large gap between a Django app that works on your laptop and a Software-as-a-Service product that strangers pay to use every day. The tutorials get you across the first gap — you can build a blog, a to-do list, a CRUD app. This book is about the second gap: multi-tenancy, subscription billing, background work, transactional email, authorization, security, deployment, and the operational discipline that keeps a paying product running. We will cross that gap by building one real application from an empty directory to a deployed, billable SaaS, and every chapter adds a layer that a production product actually needs.

## What we are building

Throughout the book we build TaskFlow, a multi-tenant team task-management SaaS. It is deliberately a familiar kind of product — organizations sign up, invite teammates, create projects, and track tasks — because the point is not the novelty of the domain but the machinery around it. That machinery is identical whether you are building task management, a CRM, an analytics dashboard, or an invoicing tool: many isolated customers sharing one deployment, each paying a recurring fee, each with their own users and data that must never leak into another's. By the final chapter TaskFlow will have organizations with isolated data, users with roles, a working subscription with Stripe billing and a customer portal, background jobs sending email and generating exports, a hardened production configuration, and a repeatable deployment. You will be able to lift any one of these subsystems into your own

product almost unchanged.

## Who this book is for

This book assumes you know Django — models, views, templates, the ORM, migrations — at the level of having built at least one small app. You do not need prior experience with multi-tenancy, Stripe, Celery, or Docker; each is introduced where we need it. If you have ever finished a Django tutorial and thought "but how do I actually turn this into a product people pay for?", this book is the answer to that question, worked end to end rather than as disconnected topics.

## The anatomy of a SaaS

Before writing any code it helps to name the parts every subscription product shares, because they recur in the same shape across almost all of them. Seeing them as a set turns "build a SaaS" from a vague ambition into a concrete checklist. Multi-tenancy is the defining trait: one running application serves many customers (tenants) whose data is strictly isolated. Getting this boundary right is the single most important architectural decision, because a leak across it is a catastrophic breach of trust. Accounts and authentication let users prove who they are and belong to a tenant. Authorization decides what each user may do within their tenant, usually through roles. Billing turns usage into recurring revenue and is a state machine of its own, driven by an external payment processor. Background work handles everything that should not happen inside a web request — sending email, generating reports, running scheduled jobs. Transactional email carries invitations, receipts, and notifications reliably. And wrapping all of it, operations — deployment, security, monitoring, backups — is what makes the difference between a demo and a product. Each of these is a chapter or more in this book. None is optional in a real product, and the mistake most first SaaS attempts make is treating the operational and billing layers as afterthoughts to bolt on later, when in fact they shape the architecture from the start.

## The architecture we will use

We will build TaskFlow with a light but deliberate architecture rather than Django's default of logic-in-views-and-models. Business operations live in a service layer — plain functions that encapsulate a use case like "create an organization" or "invite a member" — and read logic lives in selectors. Views stay thin, translating HTTP to service calls and back. This is not enterprise ceremony; it is the smallest amount of structure that keeps business logic findable and

testable as the app grows past a handful of features, and it pays off enormously once billing, background jobs, and a CLI all need to invoke the same operations. We introduce this pattern in Chapter 3 and use it consistently thereafter, so by the end you will have internalized a way of organizing Django code that scales from a side project to a team codebase.

## The technology stack

TaskFlow is built on current, boring, production-proven technology, because a product's job is to be reliable, not to showcase the newest tool. We use Django 5.2 on Python 3.12+, PostgreSQL as the database (never SQLite in production — we rely on Postgres features and its concurrency behavior), and Redis as cache and message broker. Background jobs run on Celery. Billing is handled by Stripe. The frontend is server-rendered templates enhanced with HTMX and a little Alpine.js for interactivity without a separate single-page-app build. Deployment uses Docker, Gunicorn, and Nginx. Every one of these choices is explained where it first appears. We deliberately avoid a heavy JavaScript frontend. For the overwhelming majority of SaaS products a server-rendered app with HTMX delivers the interactivity users expect at a fraction of the complexity and team cost of a decoupled SPA, and it keeps this book focused on the backend concerns that actually make a SaaS work.

## How the book is structured

The chapters follow the order you would actually build in. We set up the project and its configuration, model the domain with a clean architecture, add authentication and multi-tenancy, build the core product, wire up billing and background jobs, layer on authorization and a real frontend, then turn to testing, security, deployment, and operations before a launch checklist. Each chapter ends with a working, runnable increment — you are never left with code that does not yet do anything. You can read straight through and build TaskFlow alongside, or treat individual chapters as references when you need to add a subsystem to your own product. The billing chapter stands alone well; so do the multi-tenancy, background-jobs, and deployment chapters. But the greatest value is in seeing how the pieces fit, because the hard part of a SaaS is rarely any single subsystem — it is the interactions between them, and those only become clear when you build the whole thing.

## A tour of what you will build

It helps to picture the finished product before we start assembling it. When TaskFlow is complete, a visitor lands on a marketing page and signs up with their email. They verify the

address, are prompted to create an organization, and become its owner. Inside, they create projects and tasks, invite colleagues by email, and assign work. Each invited colleague joins the same organization and sees the same data — but users of any other organization see none of it. When the owner is ready to unlock paid features or add more seats, they hit a pricing page, check out through Stripe, and their organization's plan updates automatically as Stripe confirms the payment. Behind the scenes, invitations and receipts are emailed by background workers, scheduled jobs clean up expired data, and every action is logged and monitored. Nothing in that description is unusual — it is the baseline experience of virtually every B2B SaaS. That is the point. By building this ordinary-but-complete product properly, you learn the shape that generalizes to whatever you build next, and you finish with a codebase whose subsystems you can transplant.

## The business shape of a SaaS

A subscription product is not just a technical artifact; its business model shapes the engineering in ways worth understanding up front. Revenue is recurring, measured as monthly recurring revenue (MRR), and the enemy of recurring revenue is churn — customers who cancel or whose payments fail. This is why billing cannot be an afterthought bolted on at the end: the reliability of your webhook handling, the grace you extend on a failed payment, and the ease of managing a subscription directly affect revenue. A dropped webhook is not a cosmetic bug; it is a customer who paid but did not get access, or one who cancelled but kept it. The subscription model also changes how you think about customers. They are not one-time buyers but ongoing relationships, which is why per-tenant data isolation, reliable email, and operational uptime matter so much — a customer trusts you with their data and their recurring payment, month after month. Keeping that trust is the actual product. The features are table stakes; the reliability is the moat.

## How to read this book

The most rewarding way to use this book is to build TaskFlow alongside it, typing the code and running each increment, because the understanding that sticks comes from making it work on your own machine. Each chapter produces something runnable, so you are never more than a chapter from seeing progress. If instead you are here to add a specific capability to an existing product — billing, multi-tenancy, background jobs — the relevant chapters stand alone well enough to serve as focused references. Code in this book is shown in the context where it lives, with import lines and boilerplate trimmed once a pattern is established so the signal stays high. When a snippet omits detail for brevity, the surrounding prose says what it stands in for. The aim is always to show you enough to understand and adapt the idea,

not to be a copy-paste dump that hides the reasoning.

## The metrics that will shape your decisions

Because a SaaS lives or dies by recurring revenue, a handful of business metrics quietly drive many engineering priorities, and it helps to know them before you build. MRR (monthly recurring revenue) is the heartbeat — the predictable revenue you can count on each month. Churn is its enemy: the percentage of customers or revenue you lose each month, whether by cancellation or by failed payment. A product that adds customers faster than it loses them grows; one that does not, however good, slowly bleeds out. LTV (lifetime value) is how much a customer is worth over their whole relationship, and CAC (customer acquisition cost) is what it cost to win them; the ratio between them decides whether the business is viable. Why does this belong in an engineering book? Because these numbers explain why certain code matters more than it appears to. Reliable webhook handling protects revenue against silent churn from failed payments. A smooth onboarding reduces the early cancellations that wreck LTV. A product that stays up and keeps data safe is one customers do not leave. When we make an architectural choice in later chapters and call it "worth the effort," it is usually because it protects one of these numbers. Engineering a SaaS well is, in the end, engineering these metrics.

## What "done" looks like, subsystem by subsystem

It is easy to build the happy path of a feature and call it finished, then discover in production that "done" meant much more. Throughout this book, "done" for each subsystem includes its unglamorous edges. Authentication is not done when login works; it is done when verification, password reset, rate limiting, and account lockout work. Billing is not done when a checkout succeeds; it is done when webhooks are verified and idempotent, failed renewals are handled gracefully, and the local state reconciles with Stripe. Multi-tenancy is not done when data is scoped in the obvious views; it is done when isolation is enforced everywhere and proven by tests that try to break it. Holding this fuller definition of done is a large part of what separates a product from a demo, and each chapter is explicit about where that line sits.

## Common misconceptions to leave behind

A few beliefs make building a first SaaS harder than it needs to be, and it is worth naming them now. The first is that you need a fashionable, complex stack — a JavaScript frontend framework, microservices, Kubernetes — to build something real. You do not; a

well-structured Django monolith serves the vast majority of successful SaaS products for years, and the complexity you skip is complexity you do not have to operate. The second is that billing and operations are things to add at the end. They are not; they shape the architecture and are woven through this book from early on. The third is that scaling is an urgent, upfront concern. For almost every product it is not — you will spend far longer looking for customers than fighting load, and premature scaling is wasted effort that slows you down. We build for correctness, security, and maintainability first, and scale what measurements later tell us to.

## Production-ready means honest about trade-offs

A recurring theme is that "production-ready" is not a checklist you complete once but a series of deliberate trade-offs. There is no single right way to do multi-tenancy, to model permissions, or to structure a deployment; there are choices with different costs that suit different scales and teams. Where a decision matters, this book explains the alternatives and why we chose what we chose, so you can make a different call when your context differs. The goal is not to hand you dogma but to make you the kind of engineer who ships products that stay up, bill correctly, and keep customers' data safe. With that framing set, let us start where every real project starts: a clean, well-configured foundation. In the next chapter we create the TaskFlow project and set up the configuration that will carry it from local development all the way to production.

CHAPTER 2

# Project Setup

The foundation you pour in the first hour shapes everything built on it. A project set up casually — one settings file, dependencies installed ad hoc, secrets in the code — fights you for its entire life. A project set up deliberately, with a clean structure and a configuration that separates development from production from the start, stays pleasant to work in for years. This chapter builds that foundation for TaskFlow: the environment, the project layout, a settings architecture that scales, and the tooling that keeps quality from eroding.

## Prerequisites and environment

You need Python 3.12 or newer, PostgreSQL, and Redis available locally. We do not develop against SQLite even though Django defaults to it, because TaskFlow uses PostgreSQL-specific features and because developing on a different database than you deploy to hides bugs until production — the one place you never want to find them. Install Postgres and Redis natively or run them in Docker; we provide a development Docker Compose file later in the chapter. Create an isolated virtual environment for the project. Isolation is not optional: it keeps TaskFlow's dependencies from colliding with other projects and makes the exact set of packages reproducible.

```
python -m venv .venv
source .venv/bin/activate # Windows: .venv\Scripts\activate
python -m pip install --upgrade pip
```

## Managing dependencies

Dependencies must be pinned and reproducible. We keep a human-edited list of direct requirements and compile it into a fully pinned lock file, so that every environment — your laptop, a teammate's, CI, production — installs the identical set of packages. The simplest robust approach is `pip-tools`: you write `requirements.in` with your direct dependencies and version floors, and `pip-compile` produces a pinned `requirements.txt`.

```
# requirements.in
Django>=5.2,<6.0
psycopg[binary]>=3.2
redis>=5.0
celery>=5.4
django-allauth>=65.0
stripe>=11.0
django-environ>=0.11
gunicorn>=23.0
```

Pin to a major version with a floor and a ceiling — `>=5.2,<6.0` — so you get security and bug-fix releases automatically but never an untested major upgrade that breaks you in the middle of a deploy. Run `pip-compile requirements.in` to generate the locked file, and `pip-sync` to install exactly it.

## The pip-tools workflow in detail

The two-file split — a hand-written `requirements.in` of intent and a machine-generated `requirements.txt` of fact — is the whole idea, and it repays a closer look because most dependency pain in Django projects comes from conflating those two files. The `.in` file lists only what you chose to depend on, with the loosest constraint you are willing to accept. The `.txt` file is the compiled result: every direct package and every transitive one, pinned to an exact version, with a comment recording which parent pulled it in. You never edit the `.txt` by hand; you edit intent in the `.in` and recompile. We split further into a base file and a development overlay, because CI and production have no business installing `pytest` or the debug toolbar. The dev file references the base with `-r`, so a single compile resolves both together and guarantees the two lock files never disagree on a shared transitive version.

```
# requirements-dev.in
-r requirements.in
pytest>=8.0
pytest-django>=4.9
ruff>=0.6
pip-tools>=7.4
django-debug-toolbar>=4.4
```

Compiling both files, with hashes for supply-chain integrity, is a fixed pair of commands. The `--generate-hashes` flag records a cryptographic hash of every wheel, so a compromised or swapped package on the index is rejected at install time rather than silently trusted.

```
# Compile lock files from the .in intent files
pip-compile --generate-hashes requirements.in
pip-compile --generate-hashes requirements-dev.in
# Install EXACTLY the locked set (removes anything not
listed)
pip-sync requirements-dev.txt
```

`pip-sync` is the underrated half of the workflow. Plain `pip install -r` only adds packages; it never removes one you have since dropped from the `.in`, so environments accumulate cruft that works on your machine and fails in CI. `pip-sync` makes the environment match the lock file exactly — installing what is missing and uninstalling what is stale — which is precisely the reproducibility guarantee we are paying for. Upgrades are deliberate, never incidental. To bump a single package you name it; to refresh everything within your declared ceilings you upgrade wholesale. Either way the change lands as a reviewable diff in `requirements.txt`, so a security bump is a visible line in a pull request rather than an invisible drift.

```
# Upgrade just one dependency, respecting the ceiling in
requirements.in
pip-compile --upgrade-package django requirements.in
# Refresh every pin to the newest allowed by the constraints
pip-compile --upgrade requirements.in
```

The honest trade-off: pip-tools adds a compile step and a second file to keep in sync, and the first time a hash mismatch blocks an install it feels like friction. That friction is the feature. A lock file that can drift is not a lock file, and the fifteen seconds a recompile costs is trivial against an afternoon spent bisecting why production resolved a different version of a transitive dependency than your laptop did.

## Creating the project

Start the Django project with a structure that separates the project configuration from the applications. We use a top-level `config` package for project-wide settings and URLs, and an `apps` directory to hold our applications, which keeps the repository root uncluttered as the number of apps grows.

```
django-admin startproject config .
mkdir apps
python manage.py startapp accounts apps/accounts
```

Putting apps under a package means their import paths become `apps.accounts`, which you register accordingly in settings. This is a small convention with a large payoff in readability: at

a glance you can tell your own apps from third-party ones, and the root directory stays legible.

## The apps directory layout and rationale

TaskFlow's domain divides cleanly into a handful of apps, and drawing those boundaries early — before there is code to untangle — is one of the highest-leverage decisions in the whole setup. Each app owns one slice of the domain and exposes its behaviour through the service layer, never by reaching into another app's models directly. The layout mirrors the domain described in Chapter 1.

```
apps/
accounts/ # custom email User, authentication
organizations/ # Organization (tenant), Membership, roles
projects/ # Project, Task, and their services
billing/ # Plan, Subscription, Stripe integration
common/ # shared base models, mixins, utilities
```

Every app follows the same internal shape, and the two files that matter most are ones Django does not create for you. `services.py` holds the write operations — the plain functions that create an organization, invite a member, move a task to done — and `selectors.py` holds the reads. Keeping these separate from `models.py` is the spine of the architecture: models stay thin and describe data, views stay thin and translate HTTP, and all the business logic lives in one predictable place per app.

```
apps/projects/
__init__.py
apps.py
models.py # Project, Task — data and light invariants
services.py # create_task, move_task, assign_task — writes
selectors.py # tasks_for_project, board_for_org — reads
admin.py
urls.py
views.py # thin: parse request, call service, render
migrations/
tests/
test_services.py
test_selectors.py
```

Note that `tests` is a package, not the single `tests.py` Django scaffolds, because a real app accumulates more test code than production code and one file becomes unworkable. The

`common` app is deliberately dependency-free: it holds the abstract `TimeStampedModel`, tenant-scoping mixins, and small utilities that every other app imports, and it imports from none of them. Enforcing that one-directional dependency — feature apps may depend on `common`, never the reverse — keeps the graph acyclic and the codebase navigable as it grows. Give each app an explicit `AppConfig` with its dotted path, because the `apps/` nesting means Django cannot infer the label from the directory alone. This is the small piece of glue that makes `apps.projects` import cleanly.

**apps/projects/apps.py** · python

```python
from django.apps import AppConfig
class ProjectsConfig (AppConfig):
default_auto_field = "django.db.models.BigAutoField"
name = "apps.projects"
label = "projects"
```

## A settings architecture that scales

A single `settings.py` with `if DEBUG:` branches becomes unmanageable and dangerous — it is how a debug toolbar or a permissive setting leaks into production. Instead we split settings into a package with a shared base and per-environment overrides.

```
config/settings/
__init__.py
base.py # everything common
development.py # imports base, relaxes for local work
production.py # imports base, hardens for deploy
```

The base module holds every setting common to all environments. The environment modules import it and change only what differs. You select the environment with the `DJANGO_SETTINGS_MODULE` variable — `config.settings.development` locally, `config.settings.production` on the server — so there is no runtime branching and no chance of production accidentally running development configuration.

**config/settings/base.py** · python

```python
import environ
from pathlib import Path
BASE_DIR = Path(__file__).resolve().parent.parent.parent
env = environ.Env()
SECRET_KEY = env("DJANGO_SECRET_KEY")
INSTALLED_APPS = [
```

```
"django.contrib.admin", "django.contrib.auth",
"django.contrib.contenttypes", "django.contrib.sessions",
"django.contrib.messages", "django.contrib.staticfiles",
# third party
"allauth", "allauth.account",
# local
"apps.accounts",
]
DATABASES = {"default": env.db("DATABASE_URL")}
CACHES = {"default": env.cache("REDIS_URL",
default="redis://localhost:6379/1")}
```

## A full base.py walkthrough

The skeleton above shows the shape; a production base file carries more, and every line earns its place. Below is a fuller `base.py` with the settings TaskFlow actually relies on — grouped so that anyone reading it can see, section by section, what the project has decided. The guiding rule is that base holds decisions that are true everywhere; anything that differs between a laptop and a server is read from the environment or deferred to an overlay.

config/settings/base.py · python

```
import environ
from pathlib import Path
BASE_DIR = Path(__file__).resolve().parent.parent.parent
env = environ.Env()
# --- Core ---------------------------------------------------
------------
SECRET_KEY = env("DJANGO_SECRET_KEY")
DEBUG = False # overlays turn this on; base is safe
ROOT_URLCONF = "config.urls"
WSGI_APPLICATION = "config.wsgi.application"
DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"
# --- Applications
-------------------------------------------------------
DJANGO_APPS = [
"django.contrib.admin", "django.contrib.auth",
"django.contrib.contenttypes", "django.contrib.sessions",
"django.contrib.messages", "django.contrib.staticfiles",
]
THIRD_PARTY_APPS = ["allauth", "allauth.account"]
```

```
LOCAL_APPS = [
"apps.common", "apps.accounts", "apps.organizations",
"apps.projects", "apps.billing",
]
INSTALLED_APPS = DJANGO_APPS + THIRD_PARTY_APPS + LOCAL_APPS
MIDDLEWARE = [
"django.middleware.security.SecurityMiddleware",
"django.contrib.sessions.middleware.SessionMiddleware",
"django.middleware.common.CommonMiddleware",
"django.middleware.csrf.CsrfViewMiddleware",
"django.contrib.auth.middleware.AuthenticationMiddleware",
"django.contrib.messages.middleware.MessageMiddleware",
"django.middleware.clickjacking.XFrameOptionsMiddleware",
"allauth.account.middleware.AccountMiddleware",
]
# --- Data and cache
------------------------------------------------
```

```
DATABASES = {"default": env.db("DATABASE_URL")}
DATABASES["default"]["CONN_MAX_AGE"] =
env.int("CONN_MAX_AGE", default=60)
DATABASES["default"]["ATOMIC_REQUESTS"] = True
CACHES = {"default": env.cache("REDIS_URL",
default="redis://localhost:6379/1")}
# --- Auth ------------------------------------------------
------------
AUTH_USER_MODEL = "accounts.User" # custom email-based user,
Chapter 3
LOGIN_REDIRECT_URL = "dashboard"
AUTH_PASSWORD_VALIDATORS = [
{"NAME": "django.contrib.auth.password_validation."
"UserAttributeSimilarityValidator"},
{"NAME": "django.contrib.auth.password_validation."
"MinimumLengthValidator"},
{"NAME": "django.contrib.auth.password_validation."
"CommonPasswordValidator"},
]
# --- Templates
------------------------------------------------------------
TEMPLATES = [{
"BACKEND":
"django.template.backends.django.DjangoTemplates",
```

```
"DIRS": [BASE_DIR / "templates"],
"APP_DIRS": True ,
"OPTIONS": {"context_processors": [
"django.template.context_processors.request",
"django.contrib.auth.context_processors.auth",
"django.contrib.messages.context_processors.messages",
]},
}]
# --- i18n / static / media
-----------------------------------------
LANGUAGE_CODE = "en-us"
TIME_ZONE = "UTC"
USE_I18N = True
USE_TZ = True
STATIC_URL = "/static/"
STATIC_ROOT = BASE_DIR / "staticfiles"
MEDIA_URL = "/media/"
MEDIA_ROOT = BASE_DIR / "media"
# --- Integrations (read here, used by the service layer)
--------------
STRIPE_SECRET_KEY = env("STRIPE_SECRET_KEY")
STRIPE_WEBHOOK_SECRET = env("STRIPE_WEBHOOK_SECRET")
CELERY_BROKER_URL = env("REDIS_URL",
default="redis://localhost:6379/0")
```

Two decisions there are worth calling out because they are quiet but consequential. `ATOMIC_REQUESTS = True` wraps every request in a database transaction, so a view that raises halfway through never leaves a half-written organization or an orphaned membership — a sane default for a system where a single user action often touches several tables. And `CONN_MAX_AGE` keeps database connections alive between requests, which matters as soon as you have real traffic: opening a fresh Postgres connection per request is pure latency you can reclaim with one integer. Both belong in base because they are correct in every environment; only their tuning ever changes.

## Configuration from the environment

The single most important configuration rule is that secrets and environment-specific values never live in code. The database URL, the secret key, Stripe keys, email credentials — all come from the environment, read here with `django-environ`. In development you keep them in a `.env` file that is git-ignored; in production they are set as real environment variables by your deployment platform. This is the twelfth of the twelve-factor principles, and it is what lets the

identical code run safely in every environment while keeping secrets out of your version history.

```
# .env (git-ignored — never commit this)
DJANGO_SECRET_KEY=dev-only- not -a-real-secret
DATABASE_URL=postgres://taskflow:taskflow@localhost:5432/tas
kflow
REDIS_URL=redis://localhost:6379/0
DJANGO_SETTINGS_MODULE=config.settings.development
```

Commit a `.env.example` with the keys and dummy values so a new developer knows exactly what to set, but never the real `.env`. A leaked secret key or database password in git history is one of the most common and most damaging security mistakes, and it is entirely avoidable with this discipline from day one.

## A complete .env.example and .gitignore

The `.env.example` is documentation that cannot rot, because a missing key surfaces the moment the app fails to boot. List every variable the app reads, with a safe dummy value or a blank, and group them the way base.py groups its settings so the two files read as a pair. A developer who copies this to `.env` and fills in real secrets has a complete configuration and no guesswork.

```
# .env.example (commit this; copy to .env and fill in real
values)
# --- Core ---
DJANGO_SETTINGS_MODULE=config.settings.development
```

```
DJANGO_SECRET_KEY=change-me-to-a-50-char-random-string
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1
# --- Data ---
DATABASE_URL=postgres://taskflow:taskflow@localhost:5432/tas
kflow
REDIS_URL=redis://localhost:6379/0
CONN_MAX_AGE=60
# --- Stripe (test keys locally) ---
STRIPE_SECRET_KEY=sk_test_xxx
STRIPE_WEBHOOK_SECRET=whsec_xxx
# --- Observability (optional; blank disables) ---
SENTRY_DSN=
```

The `.gitignore` is the guardrail that makes the "never commit secrets" rule mechanical rather than a matter of memory. Exclude the environment file, the virtualenv, Python and build artifacts, local media, and editor and coverage droppings. Note that `.env` is ignored but `.env.example` is explicitly re-included — the one line that keeps your template committed while your secrets stay out.

```
# .gitignore
# Secrets and environment
.env
!.env.example
# Python
__pycache__/
*.py[cod]
.venv/
*.egg-info/
# Django
/staticfiles/
/media/
*.log
db.sqlite3
# Tooling
.pytest_cache/
.ruff_cache/
.coverage
htmlcov/
# Editors / OS
.idea/
.vscode/
```

```
.DS_Store
```

## Development versus production settings

The environment modules express the difference between "make me productive" and "keep me safe". Development turns on the debug toolbar and verbose errors; production turns them firmly off and switches on every security setting. We flesh production out fully in the deployment chapter, but the shape is clear from the start.

config/settings/development.py · python

```
from .base import * # noqa
```

```
DEBUG = True
ALLOWED_HOSTS = ["localhost", "127.0.0.1"]
INSTALLED_APPS += ["debug_toolbar"]
# config/settings/production.py
from .base import * # noqa
DEBUG = False
ALLOWED_HOSTS = env.list("DJANGO_ALLOWED_HOSTS")
SECURE_SSL_REDIRECT = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

## Setting up the database

Create the PostgreSQL database and a dedicated role for the application, granting it only what it needs. Using a scoped application role rather than a superuser is a small habit with real security value — if the application is ever compromised, the blast radius is limited to its own database.

```
CREATE ROLE taskflow WITH LOGIN PASSWORD 'taskflow';
CREATE DATABASE taskflow OWNER taskflow;
```

With the database reachable through `DATABASE_URL`, run the initial migrations and confirm everything connects.

```
python manage.py migrate
python manage.py runserver
```

## Version control and quality tooling

Initialize git immediately and add a `.gitignore` that excludes the virtual environment, the `.env` file, compiled files, and local media. Then set up the tooling that keeps code quality from drifting as the project grows: a formatter and linter ( Ruff handles both fast), and pre-commit hooks that run them automatically before each commit so unformatted or broken code never enters history.

**.pre-commit-config.yaml** · python

```
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
rev: v0.6.0
```

```
hooks:
- id: ruff
- id: ruff-format
```

Enforcing formatting and linting automatically from the first commit costs nothing and saves endless bikeshedding and messy diffs later. It is far easier to hold a standard from the start than to impose one on a large, inconsistent codebase.

## Configuring Ruff, pre-commit, and EditorConfig

Ruff is configured once in `pyproject.toml` and then every tool — your editor, the pre-commit hook, CI — reads the same rules, so there is a single source of truth for what "correct style" means. We enable a focused set of rule families rather than everything at once: pyflakes for real errors, pycodestyle for layout, isort for import order, pyupgrade to keep syntax modern, flake8-bugbear for likely bugs. A wall of noise trains people to ignore the linter, so we start strict-but-sane and add rules deliberately.

**pyproject.toml** · python

```
[tool.ruff]
line-length = 88
target-version = "py312"
[tool.ruff.lint]
select = ["E", "F", "I", "UP", "B", "DJ"] # DJ =
flake8-django
ignore = ["E501"] # formatter owns line length
[tool.ruff.lint.isort]
known-first-party = ["apps", "config"]
```

```
[tool.ruff.format]
quote-style = "double"
```

The pre-commit config gets two more hooks alongside Ruff: a whitespace and end-of-file tidier, and a guard against ever committing large binaries. Then a one-time `pre-commit install` wires the hooks into git so they run on `git commit` without anyone having to remember them.

**.pre-commit-config.yaml** · python

```
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
rev: v0.6.0
```

```
hooks:
- id: ruff
args: [--fix]
- id: ruff-format
- repo: https://github.com/pre-commit/pre-commit-hooks
rev: v4.6.0
hooks:
- id: trailing-whitespace
- id: end-of-file-fixer
- id: check-added-large-files
```

An `.editorconfig` file closes the loop for anything the linter does not touch and for teammates whose editors are configured differently. It is a tiny, universally supported file that keeps indentation and line endings consistent before Ruff ever runs — the difference between a diff that shows your change and one drowned in whitespace churn.

```
# .editorconfig
root = true
[*]
charset = utf-8
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
indent_style = space
[*.py]
indent_size = 4
[*.{html,css,js,yml,yaml,json}]
indent_size = 2
[Makefile]
```

```
indent_style = tab
```

## A development environment with Docker Compose

To give every developer an identical Postgres and Redis without manual installation, provide a Compose file for the supporting services. The application itself can still run on the host for fast reloads, while its dependencies run in containers.

```
# docker-compose.yml (development services)
services :
```

```
db :
image : postgres:16
environment : { POSTGRES_DB : taskflow, POSTGRES_USER :
taskflow, POSTGRES_PASSWORD : taskflow}
ports : ["5432:5432"]
redis :
image : redis:7
ports : ["6379:6379"]
```

A new developer now runs `docker compose up -d`, installs the pinned requirements, copies `.env.example` to `.env`, and has a working environment in minutes. That low-friction onboarding matters more than it sounds: it is the difference between a project a teammate can contribute to today and one that costs them a lost afternoon of setup.

## A production-preview Compose stack

The development Compose file runs only the backing services; the app runs on your host for fast reloads. That is the right trade-off for daily work, but it means you never exercise the real production topology — Gunicorn behind Nginx, a Celery worker, static files collected — until you deploy. A second Compose file closes that gap: it builds and runs TaskFlow exactly as production will, so you can catch a broken `collectstatic` or a missing environment variable on your laptop instead of in a deploy. This is not your deployment target; it is a faithful preview of it.

```
# docker-compose.prod.yml (production-preview — run the real
topology locally)
services :
web :
build : .
command : gunicorn config.wsgi:application --bind
0.0.0.0:8000 --workers 3
environment :
```

```
DJANGO_SETTINGS_MODULE : config.settings.production
env_file : [.env.prod]
depends_on : [db, redis]
expose : ["8000"]
worker :
build : .
command : celery -A config worker --loglevel=info
```

```
env_file : [.env.prod]
depends_on : [db, redis]
nginx :
image : nginx:1.27
volumes :
- ./deploy/nginx.conf:/etc/nginx/conf.d/default.conf:ro
- staticfiles:/app/staticfiles:ro
ports : ["8080:80"]
depends_on : [web]
db :
image : postgres:16
environment : { POSTGRES_DB : taskflow, POSTGRES_USER :
taskflow, POSTGRES_PASSWORD : taskflow}
volumes : ["pgdata:/var/lib/postgresql/data"]
redis :
image : redis:7
volumes :
pgdata :
staticfiles :
```

The honest caveat: this file is a rehearsal, not the performance. It uses a throwaway Postgres password and runs on `localhost`, so it is emphatically not something to expose to the internet. Its value is that `docker compose -f docker-compose.prod.yml up --build` forces you to run under `config.settings.production` with `DEBUG=False`, behind a real WSGI server, with a separate worker — the four things most likely to behave differently from `runserver`. We reuse this same image and layout, hardened, in the deployment chapter.

## Failing fast on misconfiguration

Configuration read from the environment has a failure mode: a missing or misspelled variable. The worst version of this bug is silent — a setting defaults to something insecure, or a feature quietly misbehaves, and you discover it in production. The fix is to fail fast: read required settings without a default so the application refuses to start if they are absent, turning a subtle runtime bug into an obvious startup error.

```
# base.py — no default means "required"; the app won't boot
without it
SECRET_KEY = env("DJANGO_SECRET_KEY")
DATABASES = {"default": env.db("DATABASE_URL")}
# Only genuinely optional settings get a default
SENTRY_DSN = env("SENTRY_DSN", default="")
```

A production process that will not start because `DATABASE_URL` is unset is doing exactly the right thing — it is far better than one that starts and then serves errors, or worse, silently connects to the wrong database. Make "required config is required" a hard rule and a whole class of deployment mishaps disappears.

## Validating configuration with a system check

Reading a variable as required catches the case where it is missing entirely, but not the case where it is present and wrong — a debug flag left on in production, a live Stripe key paired with a debug build. Django's system check framework is the right home for these cross-setting invariants, because it runs automatically on every `manage.py` command and can be gated in CI with `manage.py check --deploy --fail-level ERROR`. Register a small check in the `common` app that encodes the rules you never want violated.

apps/common/checks.py · python

```
from django.conf import settings
from django.core.checks import Error, register
@register(deploy= True )
def check_production_safety(app_configs, **kwargs):
errors = []
if not settings.DEBUG and
settings.SECRET_KEY.startswith("dev-"):
errors.append(Error(
"A development SECRET_KEY is set with DEBUG off.",
id="taskflow.E001",
))
if not settings.DEBUG and "test" in
settings.STRIPE_SECRET_KEY:
errors.append(Error(
"A Stripe TEST key is set with DEBUG off.",
id="taskflow.E002",
))
return errors
```

The `deploy=True` flag means these checks stay quiet during ordinary development — where a dev key and a test Stripe key are exactly what you want — and only fire under `--deploy`. Wiring that command into your CI pipeline and your deploy script turns "did we remember to swap the keys?" from a human checklist item into an automated gate that fails the build. That

is the whole spirit of this chapter: encode the rule once, and let the machine enforce it forever.

## Static files, media, and logging

Two storage concerns need a home in settings from the start. Static files are your CSS, JavaScript, and images — collected and served efficiently in production. Media is user-uploaded content, which in production lives in object storage rather than local disk. We configure both fully in the deployment chapter, but the settings keys belong in base now so the app is consistent across environments.

```
STATIC_URL = "/static/"
STATIC_ROOT = BASE_DIR / "staticfiles"
MEDIA_URL = "/media/"
MEDIA_ROOT = BASE_DIR / "media"
```

Logging deserves early attention too, because the day you need logs is the day it is too late to add them. A minimal structured logging configuration that writes to the console (where your deployment platform captures it) is enough to start, and you build on it in the operations chapter. What you want to avoid is discovering during an incident that your application logs nothing useful.

## A baseline logging configuration

Here is that baseline in full — enough to be genuinely useful without pretending to be a complete observability stack. Everything logs to the console, because in a containerized deployment stdout is where the platform collects logs; writing to a file inside a container just hides them. We format each line with a timestamp, level, and logger name so a log is greppable, and we give our own apps a named logger at a level we control from the environment.

```
# config/settings/base.py — logging
LOG_LEVEL = env("DJANGO_LOG_LEVEL", default="INFO")
LOGGING = {
"version": 1,
"disable_existing_loggers": False ,
"formatters": {
"verbose": {
"format": " {asctime} {levelname} {name} {message} ",
```

```
"style": "{",
},
},
"handlers": {
"console": {
"class": "logging.StreamHandler",
"formatter": "verbose",
},
},
"root": {"handlers": ["console"], "level": "WARNING"},
"loggers": {
# Our code: level from the environment, INFO by default.
"apps": {
"handlers": ["console"],
"level": LOG_LEVEL,
"propagate": False ,
},
# Surface unhandled 500s explicitly.
"django.request": {
"handlers": ["console"],
"level": "ERROR",
"propagate": False ,
},
},
}
```

The design choices matter. The root logger sits at `WARNING` so third-party libraries stay quiet unless something is wrong, while the `apps` logger runs at `INFO` so your own service-layer functions can narrate what they do — `logger.info("task moved to done", extra={"task_id": task.id})` — without drowning in framework chatter. Setting `propagate=False` on the named loggers stops each record being handled twice. And because the level comes from `DJANGO_LOG_LEVEL`, you can drop to `DEBUG` on a misbehaving server without a code change or redeploy. It is deliberately plain; the operations chapter adds JSON formatting and error aggregation once there is traffic to justify them.

## A task runner for common commands

You will run the same handful of commands hundreds of times — migrate, run the server, run tests, compile requirements. Capturing them in a `Makefile` (or a `justfile`) removes friction and documents the project's common operations in one place, which is especially valuable for onboarding.

Makefile · shell

```
run: ; python manage.py runserver
migrate: ; python manage.py migrate
test: ; pytest
lint: ; ruff check . && ruff format --check .
lock: ; pip-compile requirements. in
```

This is a small convenience with an outsized effect on daily comfort: `make test` is easier to remember and type than the full invocation, and a new contributor can read the Makefile to learn how the project is operated without asking anyone.

## Expanding the Makefile

As the project grows, the Makefile becomes the project's operational README — the honest, executable list of everything a developer does day to day. Below is the fuller version we use for TaskFlow, covering setup, the dependency workflow, quality gates, and the Docker services. A `.PHONY` declaration tells Make these are commands, not files to build, and a `help` target that prints the available commands turns the Makefile into self-documenting onboarding.

Makefile · shell

```
.PHONY : help install lock upgrade services run migrate \
```

```
makemigrations shell test lint format check clean
help : ## Show available commands
@grep -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) \
| awk 'BEGIN {FS=":.*? ## "} {printf " %-14s %s\n", $$1,
$$2}'
install : ## Sync the dev environment to the lock file
pip-sync requirements-dev.txt
pre-commit install
lock : ## Recompile lock files from the .in files
pip-compile --generate-hashes requirements.in
pip-compile --generate-hashes requirements-dev.in
upgrade : ## Upgrade every pinned dependency
pip-compile --upgrade requirements.in
pip-compile --upgrade requirements-dev.in
services : ## Start Postgres and Redis in the background
docker compose up -d
```

```
run : ## Run the development server
python manage.py runserver
migrate : ## Apply migrations
python manage.py migrate
makemigrations : ## Create migrations for model changes
python manage.py makemigrations
shell : ## Open a Django shell
python manage.py shell
test : ## Run the test suite
pytest
lint : ## Check style without changing files
ruff check . && ruff format --check .
format : ## Auto-fix and format the code
ruff check --fix . && ruff format .
check : ## Run Django's deploy-time system checks
python manage.py check --deploy --fail-level ERROR
clean : ## Remove caches and build artifacts
find . -type d -name __pycache__ -exec rm -rf {} +
rm -rf .pytest_cache .ruff_cache htmlcov .coverage
```

One caution worth stating plainly: a Makefile is glue, not logic. Each target should be a line or two that shells out to the real tool; the moment a target grows conditionals and loops it wants to be a management command or a script instead. Kept thin, the Makefile stays readable and every command in it is one a human could also run by hand — which is exactly what you want when something breaks and you need to step through it.

## Troubleshooting a fresh setup

A first-run checklist saves the frustrating hour that every new contributor otherwise loses to the same handful of errors. These are the ones TaskFlow's setup provokes most often, each with the message you will actually see and the cause behind it.

```
django.core.exceptions.ImproperlyConfigured: Set the
DJANGO_SECRET_KEY
```

`environment variable` — or the equivalent from django-environ. This is fail-fast working as designed: your `.env` is missing or was not loaded. Confirm the file exists (you copied it from `.env.example`), that `DJANGO_SETTINGS_MODULE` points at `config.settings.development`, and that the shell reading it is the one running the command. `psycopg.OperationalError: connection refused` or `could not translate host name "db"` — Postgres is not reachable. If you run services in Docker, `docker compose up -d` and check `docker compose ps` shows `db`

healthy. A subtle variant: inside a container the host is the service name `db`, but from your host machine it is `localhost`, so a `DATABASE_URL` copied from the wrong context points at a name that does not resolve. `FATAL: password authentication failed for user "taskflow"` — the role exists but the password in `DATABASE_URL` disagrees with the one in Postgres. Recreate the role, or align the URL; do not paper over it by connecting as the Postgres superuser, which defeats the scoped-role habit we set up deliberately. `redis.exceptions.ConnectionError: Error connecting to localhost:6379` — Redis is not running, or a stray process holds the port. Start the Redis service and verify with `redis-cli ping` returning `PONG`. The cache falls back to a default URL in development, but Celery does not, so a missing Redis surfaces the moment you touch a background task.

`ModuleNotFoundError: No module named 'apps.projects'` — the nested `apps/` layout tripping over a missing `AppConfig` or a stale `INSTALLED_APPS` entry. Confirm each app's `apps.py` sets `name = "apps.projects"` with an explicit `label`, and that the repository root (not `apps/`) is on the Python path. Ruff or pre-commit "fails" on your first commit — this is not a failure, it is the hooks doing their job. `ruff-format` and the whitespace hooks rewrite files in place and stop the commit so you can review the changes; simply `git add` the fixes and commit again. If pre-commit itself is missing, you skipped `pre-commit install` after `make install`. The pattern across all of these is the same one this chapter has argued for throughout: the setup is built to fail loudly and early, with a message that names the cause, rather than to limp along and mislead you later. An error at `runserver` is a gift compared to the same misconfiguration discovered in production.

## Where we are

TaskFlow now has a clean project structure, a settings architecture that separates environments safely, configuration driven entirely from the environment, a real PostgreSQL database, pinned reproducible dependencies, and automated quality tooling. This is the unglamorous groundwork that makes everything after it easier, and it is exactly the part that hurried projects skip and regret. With the foundation solid, the next chapter turns to the heart of the application: modeling TaskFlow's domain and establishing the service-layer architecture we will use throughout.

## Summary

This chapter poured the foundation TaskFlow is built on: an isolated Python 3.12 environment, a reproducible dependency workflow with pip-tools compiling intent into fully pinned lock files, and a project layout that separates a `config` package from an `apps` directory whose every app carries its own services and selectors. We split settings into a base plus

environment overlays so production can never accidentally run development configuration, drove every secret and environment-specific value from the environment through `django-environ`, and made missing configuration a loud startup failure rather than a silent runtime one. Around the code we wrapped the tooling that keeps quality from drifting — Ruff, pre-commit, EditorConfig, a system check that guards production invariants, a baseline structured logging config, Docker Compose for both development services and a production preview, and a self-documenting Makefile. The through-line is the book's recurring theme: production-ready means being honest about trade-offs and encoding your decisions so a machine, not a memory, enforces them.

## Key takeaways

- Develop on production's database. PostgreSQL locally, never SQLite, so database-specific bugs surface on your laptop instead of in production.
- Separate intent from lock. Hand-edit `requirements.in`, compile it with pip-tools, and use `pip-sync` so every environment installs the identical, exhaustively pinned set of packages.
- Split settings, don't branch them. A base module plus per-environment overlays selected by `DJANGO_SETTINGS_MODULE` removes runtime `if DEBUG` logic and the leaks it causes.
- Secrets live in the environment, never in code. Read them with `django-environ`, keep a committed `.env.example` template, and git-ignore the real `.env`.
- Fail fast and validate. Read required settings without defaults, and add deploy-time system checks so misconfiguration stops the app at startup rather than misbehaving in production.
- Draw app boundaries early. One domain slice per app under `apps/`, each exposing behaviour through `services.py` and `selectors.py`, with a dependency-free `common` app.
- Automate quality from the first commit. Ruff, pre-commit hooks, and EditorConfig enforce a standard mechanically instead of relying on discipline and review.
- Make onboarding a few commands. Docker Compose for services, a production-preview stack, and a self-documenting Makefile turn a lost setup afternoon into minutes.

# End of sample

You have read the opening chapters of

## Build & Ship a Production SaaS with Django

The complete book continues from here — every chapter, every appendix, in PDF and EPUB.

**EUR 39.00**

**Get the full book**

https://djangozen.com/ebooks/book/build-ship-production-saas-django/