
Wartbare
Django-Projekte
Architektur und Best Practices
dj
friendlybytes

Einleitung: Warum gute Django-Projekte kippen

Fast jedes Django-Projekt beginnt mit einem guten Gefühl.
Die ersten Features sind schnell umgesetzt, der Code wirkt übersichtlich, das Team ist produktiv. Django vermittelt früh das Gefühl, alles im Griff zu haben.

Doch viele dieser Projekte kippen leise.
Nicht durch einen großen Fehler, sondern durch viele kleine Entscheidungen, die zunächst harmlos wirken.

Ein zusätzlicher Methodenaufruf im Model.
Ein bisschen Logik in der View, „nur für diesen einen Sonderfall".
Ein Utility-Modul, weil man gerade keine bessere Idee hat.

Monate später ist das Projekt schwer zu ändern. Neue Features brauchen unverhältnismäßig lange, Bugfixes erzeugen Seiteneffekte, und Refactoring wird vermieden, weil niemand sicher ist, was dabei kaputtgeht.

Das ist kein Zeichen mangelnder Kompetenz.
Im Gegenteil: Viele dieser Projekte werden von erfahrenen Entwickler:innen gebaut.

Das eigentliche Problem ist strukturell.
Django erlaubt es, sehr lange ohne explizite Architekturentscheidungen auszukommen. Diese Freiheit beschleunigt den Start – verzögert aber oft die Zukunft.

Dieses E-Book richtet sich an Entwickler:innen, die diesen Punkt erreicht haben.
Die merken, dass Django „funktioniert", aber nicht mehr trägt.

Es geht nicht darum, Django neu zu erfinden.
Sondern darum, bewusst zu entscheiden, **wo** Logik lebt, **wer** Verantwortung trägt und **wie** ein Projekt wachsen darf, ohne im Chaos zu enden.

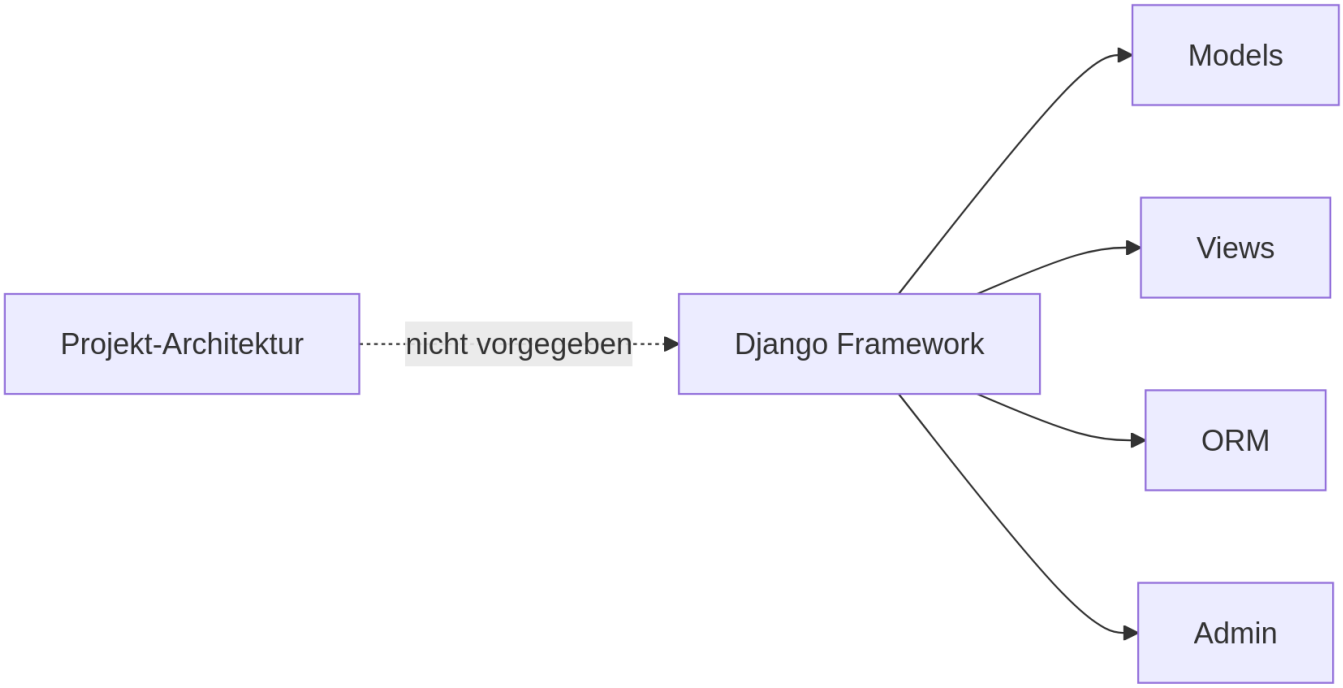
Architektur ist kein Selbstzweck.
Sie ist das Mittel, um Veränderungen wieder möglich zu machen.

Django ist kein Architektur-Framework

Django bringt viele Entscheidungen mit.
Models, Views, Apps, Admin, ORM – all das ist klar definiert und gut dokumentiert. Genau deshalb entsteht schnell der Eindruck, Django würde auch die Architektur eines Projekts vorgeben.

Das tut es nicht.

Django beschreibt Bausteine, aber keine Verantwortlichkeiten.
Es sagt dir, **wie** du ein Model definierst, aber nicht, **was** dort hingehört.
Es zeigt dir, **wie** eine View aussieht, aber nicht, **wie viel Logik** sie tragen darf.



Diese Offenheit ist bewusst gewählt.
Django will kein Dogma sein, sondern ein Werkzeugkasten.

In kleinen Projekten ist das ein Vorteil.
In wachsenden Anwendungen wird es zur Falle.

Ohne explizite Architekturentscheidungen passiert Folgendes:

- Logik wandert dorthin, wo sie gerade gebraucht wird
- technische Grenzen verschwimmen
- Konventionen entstehen zufällig, nicht geplant

Viele Teams verwechseln Django-Konventionen mit Architektur:

- „Logik ins Model“
- „Views dürfen alles, solange es funktioniert“
- „Das ORM ist unsere Domain“

Das sind Abkürzungen, keine Konzepte.

Architektur beginnt dort, wo man Django bewusst begrenzt.
Wo man entscheidet, was **nicht** in Views, Models oder Signalen passieren darf.

Django bietet die Freiheit dazu.
Wartbarkeit entsteht erst, wenn man sie nutzt.

Typische Symptome unwartbarer Django-Projekte

Unwartbarkeit entsteht selten plötzlich.
Sie zeigt sich zuerst in kleinen, wiederkehrenden Schmerzen.

Ein häufiges Symptom sind überladene Models.
Sie enthalten nicht nur Daten und einfache Regeln, sondern komplette Geschäftsprozesse, Zugriffe auf externe Services und implizite Seiteneffekte beim Speichern. Änderungen an einer Stelle wirken sich unvorhersehbar auf andere Bereiche aus.

Ein weiteres Warnsignal sind Views, die immer länger werden.
Was als einfache Request-Response-Logik begann, enthält plötzlich Validierung, Berechtigungen, Transaktionen, Sonderfälle und Feature-Flags. Die View wird zur Mini-Applikation mit eigener Logik.

Hinzu kommen versteckte Abhängigkeiten.
Signale, die „magisch" ausgelöst werden, globale Utilities, die überall genutzt werden, und Settings, die implizit Verhalten steuern. Das System funktioniert – aber niemand kann mehr erklären, warum.

Auch Tests spiegeln das Problem wider.
Entweder fehlen sie ganz, oder sie sind schwer verständlich, langsam und brechen bei kleinsten Änderungen. Tests geben keine Sicherheit, sondern verstärken die Angst vor Refactoring.

All diese Symptome haben eine gemeinsame Ursache:
Verantwortlichkeiten sind nicht klar definiert.

Wenn unklar ist, **wo** Entscheidungen getroffen werden dürfen, landen sie überall.
Das Projekt wird nicht komplex, weil es viel kann – sondern weil es alles überall kann.

Unwartbarkeit ist kein Qualitätsurteil über den Code.
Sie ist ein strukturelles Signal, das zu lange ignoriert wurde.

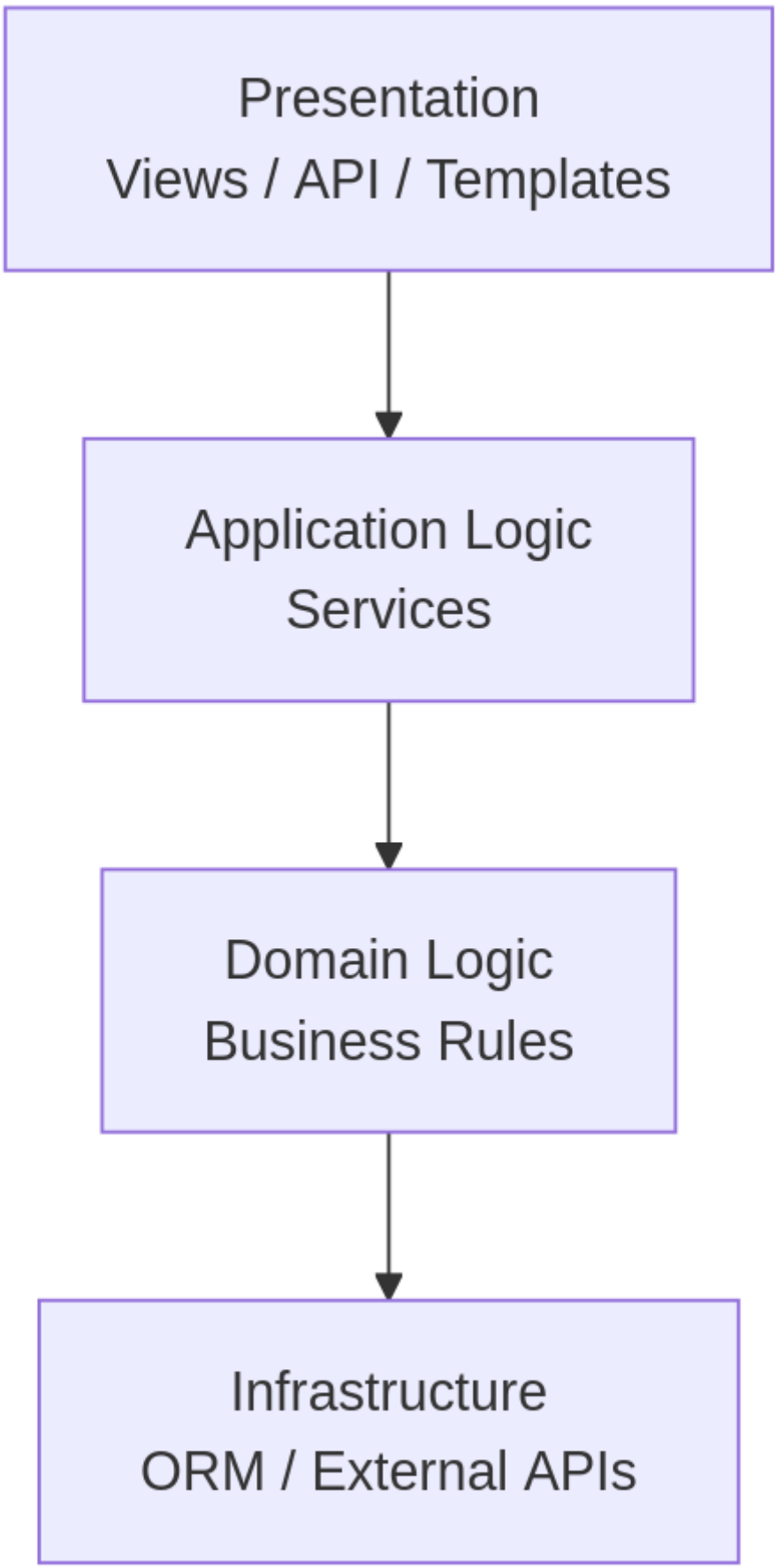
Denken in Schichten statt Dateien

Viele Django-Projekte sind nach Dateien organisiert.
Models hier, Views dort, Utilities irgendwo dazwischen. Diese Struktur folgt dem Framework – nicht dem Problem.

Wartbare Projekte denken anders.
Sie trennen nicht nach Dateitypen, sondern nach Verantwortung.

Bewährt hat sich die klare Unterscheidung von Schichten:

- Präsentation (HTTP, API, Templates)
- Anwendungslogik (Orchestrierung, Use-Cases)
- Domänenlogik (fachliche Regeln, Entscheidungen)
- Infrastruktur (ORM, externe Services, APIs)



Diese Trennung ist konzeptionell, nicht technisch.
Sie kann in einem Modul existieren oder über mehrere Apps verteilt sein. Entscheidend ist, dass jede Schicht weiß, wofür sie zuständig ist – und wofür nicht.

Eine View gehört zur Präsentationsschicht.
Sie darf Requests lesen und Responses bauen, aber keine fachlichen Entscheidungen treffen.

Die Domänenlogik kennt keine HTTP-Requests.
Sie arbeitet mit Daten und Regeln, nicht mit Frameworks.

Infrastruktur ist austauschbar.
Wenn Datenbank oder API wechseln, sollte die Kernlogik stabil bleiben.

Dieses Denken in Schichten reduziert Komplexität.
Nicht, weil es weniger Code gibt, sondern weil Veränderungen lokal bleiben.

Struktur entsteht nicht durch Ordner.
Sie entsteht durch klare Grenzen.

Die Rolle von Views richtig verstehen

In vielen Django-Projekten sind Views der Ort, an dem alles zusammenläuft.
Sie validieren Daten, prüfen Berechtigungen, verändern den Zustand des Systems und entscheiden über den weiteren Ablauf.

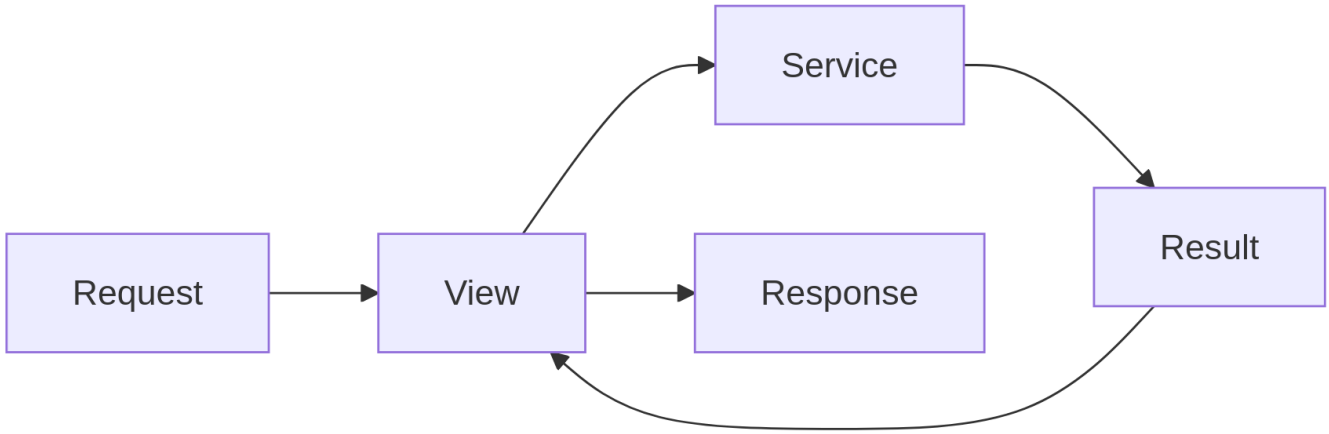
Das fühlt sich natürlich an – ist aber einer der Hauptgründe für Unwartbarkeit.

Eine View sollte kein Entscheidungszentrum sein.
Ihre Aufgabe ist es, einen Anwendungsfall zu starten und das Ergebnis darzustellen.

Wartbare Views übernehmen typischerweise:

- Entgegennahme und Vorverarbeitung des Requests
- Aufruf eines klar definierten Services
- Übersetzung des Ergebnisses in eine Response

Alles, was darüber hinausgeht, gehört nicht in die View.



Wenn fachliche Entscheidungen in Views getroffen werden, entsteht eine enge Kopplung an HTTP.
Die Logik lässt sich dann weder sinnvoll testen noch außerhalb dieses Kontextes wiederverwenden.

Dünne Views sind kein Zeichen von Einfachheit, sondern von Reife.
Sie zeigen, dass das Projekt klare Orte für Logik geschaffen hat.

Eine gute Faustregel lautet:
Wenn eine View schwer zu testen ist, enthält sie zu viel Verantwortung.

Views sollen führen, nicht denken.

Services, Selectors und klare Verantwortung

Wartbare Django-Projekte haben einen klaren Ort für Geschäftslogik.
Nicht implizit, nicht verteilt – sondern bewusst benannt.

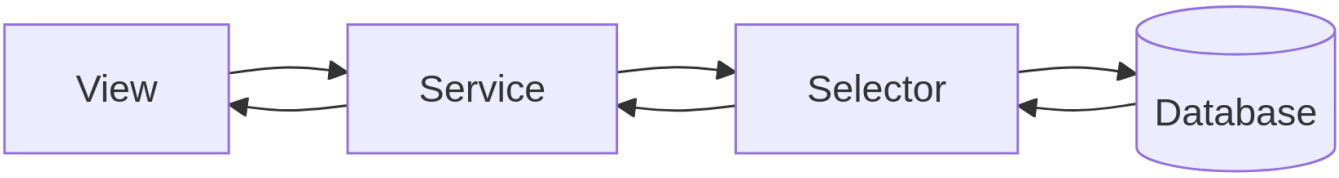
Services bilden das Herz der Anwendung.
Sie kapseln Anwendungsfälle wie „Bestellung anlegen“, „Rechnung freigeben“ oder „Benutzer deaktivieren“. Ein Service beschreibt **was** passiert, nicht **wie** es technisch umgesetzt wird.

Dadurch entstehen klare Vorteile:

- Logik ist unabhängig von HTTP
- Services sind leicht testbar
- Änderungen bleiben lokal begrenzt

Selectors erfüllen eine andere Aufgabe.
Sie sind zuständig für das Lesen von Daten – und nur dafür. Komplexe ORM-Queries, Optimierungen und Filterlogik gehören hierhin, nicht in Views oder Services.

Diese Trennung wirkt zunächst ungewohnt.
Sie verhindert aber, dass Datenzugriff und Geschäftsentscheidungen miteinander vermischt werden.



Wichtig ist die klare Grenze:

- Services dürfen Selectors nutzen
- Selectors enthalten keine Logik
- Models bleiben Datenstrukturen mit einfachen Regeln

Je expliziter diese Rollen sind, desto weniger Überraschungen gibt es im Code.
Verantwortung sichtbar zu machen ist der erste Schritt zu Wartbarkeit.

Tests als Architektur-Werkzeug

In vielen Django-Projekten werden Tests als Pflichtübung verstanden.
Man schreibt sie, weil man es sollte – nicht, weil sie wirklich helfen.

Wartbare Projekte sehen Tests anders.
Sie nutzen Tests als Feedback für die Architektur.

Wenn Logik sauber in Services gekapselt ist, lassen sich Tests einfach schreiben.
Wenn Tests dagegen kompliziert, langsam oder fragil sind, ist das oft ein Hinweis auf vermischte

Verantwortlichkeiten.

Gute Tests prüfen Verhalten, nicht Implementierungsdetails.
Sie beantworten die Frage: „Was garantiert dieses System?“ – nicht „Wie ist es intern gebaut?“

Besonders wertvoll sind Tests auf Anwendungs- und Domänenebene.
Sie kommen mit wenig Infrastruktur aus und bleiben stabil, auch wenn sich technische Details ändern.

Integrationstests haben ihren Platz.
Aber wenn jede Änderung eine komplette Datenbank, Fixtures und komplexe Setups erfordert, wird Testen zum Hindernis.

Tests schaffen Vertrauen.
Und Vertrauen ist die Voraussetzung dafür, ein System weiterzuentwickeln, ohne Angst vor Nebenwirkungen zu haben.

Projektstruktur für Wachstum

Viele Django-Projekte sind für den aktuellen Stand optimiert.
Sie funktionieren gut, solange das Team klein bleibt und die Anforderungen überschaubar sind.

Wartbare Projekte denken weiter.
Sie gehen davon aus, dass neue Features, neue Entwickler:innen und neue Anforderungen kommen werden.

Deshalb folgt ihre Struktur der Domäne, nicht der Technik.
Statt einer App für „views“ und einer für „utils“ entstehen fachlich geschnittene Bereiche wie `orders`, `billing` oder `users`.

Innerhalb dieser Bereiche liegen:

- Models
- Services
- Selectors
- Tests

Alles, was zu einer fachlichen Verantwortung gehört, bleibt zusammen.
Das reduziert mentale Sprünge und macht Änderungen nachvollziehbar.

Globale Sammelstellen werden vermieden.
Eine `utils.py`, die von überall genutzt wird, ist meist ein Zeichen fehlender Struktur.

Wachstum bedeutet nicht mehr Code.
Es bedeutet mehr Klarheit darüber, wo Code hingehört – und wo nicht.

Refactoring ohne Big Rewrite

Viele Teams erkennen irgendwann, dass ihr Django-Projekt unwartbar geworden ist.
Der reflexartige Gedanke lautet dann: „Wir müssen alles neu bauen.“

In den meisten Fällen ist das ein Fehler.

Big Rewrites sind riskant, teuer und selten erfolgreich.
Sie verschieben das Problem, statt es zu lösen.

Wartbare Projekte verbessern sich schrittweise.
Sie ziehen Struktur nach, während das System weiterläuft.

Typische erste Schritte sind:

- Logik aus Views in Services verschieben
- komplexe Queries bündeln
- implizite Seiteneffekte sichtbar machen
- Tests an kritischen Stellen ergänzen

Diese Änderungen sind klein, aber wirkungsvoll.
Sie schaffen Vertrauen und reduzieren Abhängigkeiten.



Refactoring ist kein einmaliges Ereignis.
Es ist ein kontinuierlicher Prozess, der mit jedem klar gezogenen Verantwortungsbereich leichter wird.

Nicht Perfektion macht ein Projekt wartbar.
Sondern die Fähigkeit, sich kontrolliert zu verändern.

Checkliste & nächste Schritte

Wartbarkeit ist kein Zustand, den man einmal erreicht.
Sie ist das Ergebnis fortlaufender, bewusster Entscheidungen.

Die folgende Checkliste hilft, den eigenen Projektstatus ehrlich einzuschätzen:

- Sind Views frei von Geschäftslogik?
- Gibt es klar benannte Services für zentrale Anwendungsfälle?
- Liegen komplexe Datenbankzugriffe an einem definierten Ort?
- Sind Models auf Daten und einfache Regeln beschränkt?
- Testen Tests Verhalten statt Implementierung?
- Ist die Projektstruktur fachlich und nicht technisch motiviert?
- Können Änderungen vorgenommen werden, ohne Angst vor Seiteneffekten?

Wenn mehrere dieser Fragen mit „nein“ beantwortet werden, ist das kein Scheitern.
Es ist ein Signal, dass Struktur nachgezogen werden sollte.

Der nächste Schritt muss kein radikaler Umbau sein.
Oft reicht es, bei neuen Features bewusster vorzugehen als bisher.

Architektur entsteht nicht auf dem Whiteboard.
Sie entsteht im Alltag – durch klare Verantwortung und konsequente Entscheidungen.

Wer diese Prinzipien verinnerlicht,
nutzt Django nicht nur als Framework, sondern als stabile Grundlage für langlebige Software.