

Building a User Defined Fields (UDF) System in Django

Architecture, Trade-offs, and Implementation Patterns

~45 minutes

What We'll Cover Today

GOALS

- Build a production-ready UDF system from scratch
- Understand the problems we're solving at each step
- Explore alternative paths — why some fail, why others work
- Walk away with practical, reusable knowledge

PREREQUISITES

- Basic Python and Django (models, forms, views)
- SQL databases and basic querying
- The concept of ORM (Object-Relational Mapping)

Duration: ~45 minutes

Format: Concepts + Code + Live Demo

Based on real-world implementation at Infino Legal

The Real-World Story: How We Got Here

PHASE 1

"One Size Fits All"

All clients forced into standard data model. Bankruptcy firm uses same form as insurance company.

"Where do I put the Filing Date?"

"I need to track the Claim Number!"

PHASE 2

"Custom Columns"

```
ALTER TABLE cases
ADD COLUMN filing_date DATE;
```

Problems: Form bloat, per-client required fields impossible, migrations for each change

PHASE 3

User Defined Fields

~20 semi-custom fields on a form became unmanageable. We needed a better solution.

✓ Clients configure their own fields

✓ No dev involvement needed

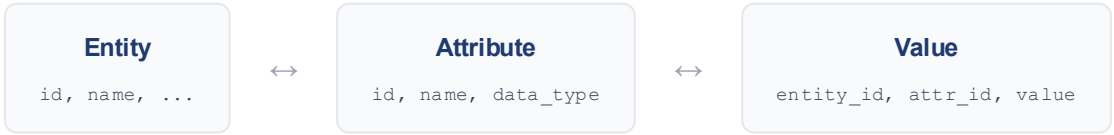
✓ Per-client required fields

The progression mirrors many growing SaaS products

PART 1

The Problem

Traditional Approach: Entity-Attribute-Value (EAV)



Query Complexity

SQL queries become unreadable. Need one JOIN per field or complex pivot queries.

Type Safety Lost

Values table stores everything as text. No DB-level type enforcement.

Poor Index Utilization

Standard indexes don't help. Queries often require full table scans.

Three tables: Entity, Attribute, Value — the classic EAV anti-pattern

EAV Query Complexity

NAIVE: ONE JOIN PER FIELD

```sql
-- Doesn't scale!
SELECT
e.id AS case_id,
v1.value AS priority,
v2.value AS incident_date
FROM entity e
LEFT JOIN value v1 ON ...
LEFT JOIN value v2 ON ...
LEFT JOIN value v3 ON ...
```

BETTER: PIVOT QUERY

```sql
-- Single JOIN, but complex
SELECT
e.id,
MAX(CASE WHEN a.name =
'priority' THEN v.value END)
AS priority,
...
GROUP BY e.id;
```

DJANGO ORM: STILL PAINFUL

```python
# One Subquery per field
Case.objects.annotate(
priority=Subquery(
Value.objects.filter(
entity_id=OuterRef('id'),
attribute__name='priority'
).values('value')[:1]
),
# ... repeat for each field
)
```

Even with pivot queries, you still have type safety and index issues

The Modern Solution: Postgres HStore / JSONB

The Concept: Semi-Structured Data

Strict SQL for core data, NoSQL document storage for custom bits. Best of both worlds.

TRADITIONAL: 500 COLUMNS

```
id | title | priority | date | ...
---+-------+----------+------+----
1  | Smith | high     | NULL | NULL

-- 450 NULL values per row
-- Massive table bloat
```

HSTORE: SINGLE COLUMN

```
id | title | user_defined_fields
---+-------+--------------------------
1  | Smith | priority=>high,
|       | incident_date=>2023-10-15

-- No NULLs, no bloat
-- Stores only what's needed
```

Our Choice: HStore (Key-Value store) — with caveats we'll discuss

HStore is like a Python dict stored in a single database column

PART 2

The Solution Architecture

Understanding HStore: What Is It Really?

Definition

HStore is a Postgres extension that stores key-value pairs in a single column using a compact binary representation.

Key Constraint

Both keys AND values are **always strings**. No nested structures, no arrays, no native types.

Storage Format

Internally uses length-prefixed strings in a flat array. Very efficient for simple key-value lookups.

SQL EXAMPLES

```sql
-- Enabling the extension
CREATE EXTENSION hstore;

-- Creating a table with HStore
CREATE TABLE cases (
id SERIAL PRIMARY KEY,
title VARCHAR(255),
user_defined_fields HSTORE
);

-- Inserting data
INSERT INTO cases (title, user_defined_fields)
VALUES (
'Smith Bankruptcy',
'priority => "high", incident_date => "2023-10-15"'
);
```

HStore is essentially a Python dict stored in a single database column

HStore vs. JSONB — Making the Choice

Feature	HStore	JSONB
Type Preservation	No (everything is string)	Yes (true stays boolean)
Nesting	No	Yes
Storage Size	Smaller for flat data	Larger (type metadata)
Index Types	GIN, GiST	GIN, GiST, btree
Django Support	HStoreField	JSONField

Why We Chose HStore

- Started on Postgres 12 with slight indexing advantages
- Flat structure (Dict[str, str]) was intentional
- Didn't want nested trees in UDFs

Recommendation for New Projects

- Use JSONB — modern Postgres (14+) has excellent indexing
- Type preservation = less work in SerDe layer
- Ecosystem standard, better tooling

The architecture patterns in this talk apply to both HStore and JSONB

Understanding GIN Indexes

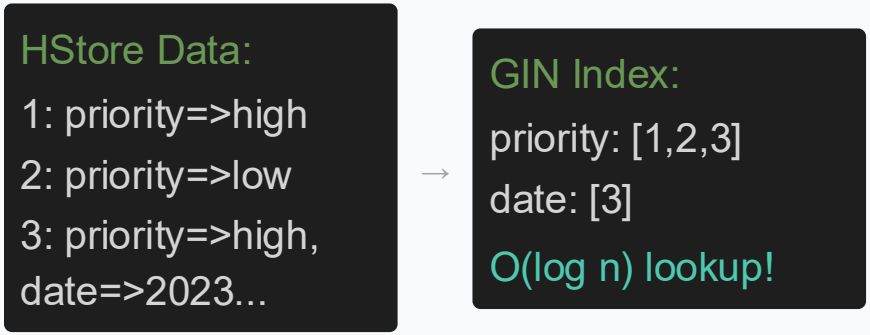
The Problem

Without an index, "find all cases where priority = high" requires a full table scan — reading every row.

GIN = Generalized Inverted Index

Like an index at the back of a book. Instead of reading every page, look up the term and get page numbers.

How GIN Works for HStore



Trade-offs: Faster reads O(log n), slightly slower writes (index maintenance), extra storage

GIN is specifically designed for "contains" queries — perfect for key-value stores

CREATING A GIN INDEX

```
-- Without this, every query scans ALL rows
CREATE INDEX idx_cases_udf
ON cases USING GIN
(user_defined_fields);

-- Now these queries are fast:
SELECT * FROM cases
WHERE user_defined_fields
? 'priority'; -- Has key?

SELECT * FROM cases
WHERE user_defined_fields
@> 'priority => high';
-- Has key=value?
```

Django's ContentType Framework

What is ContentType?

Django's built-in registry of all models in your project. Like a phone book for models.

Why We Need It

We want ONE UDFDefinition table for any model (Case, Contact, Invoice), not separate tables for each.

One Table Serving Multiple Entity Types

```
UDFDefinition:
1: Case - Priority
2: Case - Incident Date
3: Contact - Shoe Size
4: Invoice - PO Number
```

HOW CONTENTTYPE WORKS

```python
from django.contrib.contenttypes.models
import ContentType

# Get the ContentType for our Case model
case_type = ContentType.objects.get_for_model(Case)

# Returns:
# ContentType(app_label='cases',
#             model='case', id=15)

# Later, get the model class back
model_class = case_type.model_class()
# Returns: cases.models.Case
```

Write ONE generic system, apply it to any model in your project

Designing the Schema: UDFDefinition

The Key Insight

Even though **storage** is schema-less (HStore), the **application** needs a schema to know how to render and validate fields.

Why SQL Table (not JSON Schema)?

- ORM filtering (e.g., all DATE fields)
- Foreign keys for constraints
- Admin interface for free

```python
# UdfType Enum
class UdfType(models.TextChoices):
TEXT = "TEXT", "Text"
INTEGER = "INTEGER", "Integer"
DECIMAL = "DECIMAL", "Decimal"
DATE = "DATE", "Date"
BOOLEAN = "BOOLEAN", "Yes/No"
CHOICE = "CHOICE", "Dropdown"
CONTACT = "CONTACT", "Contact Ref"
```

This table drives the UI, validation, and API exports

THE SCHEMA MODEL

```python
class UDFDefinition(models.Model):
content_type = models.ForeignKey(
"contenttypes.ContentType",
on_delete=models.CASCADE,
help_text="Which model (Case, Contact...)"
)
field_type = models.CharField(
max_length=20,
choices=UdfType.choices,
help_text="Validation and widget"
)
label = models.CharField(max_length=64)
slug = models.SlugField(
max_length=50, unique=True,
help_text="Immutable key in HStore"
)
is_required = models.BooleanField(
default=False
)
```

The Naming Problem: Human vs. Machine

The Challenge

Users want to name fields "Date". Two users might both create a "Date" field. Or one user deletes "Date" and creates a new one.

Collision: If we use slugify("Date") → date as the key, we have collisions.

Solution: Randomized Slugs

`"Date" → date_x9z2a`

`"Date" (later) → date_p2q1m`

Immutable, unique keys that never conflict.

Why Not UUID?

`a1b2c3d4-e5f6-7890-abcd...` ✗

`incident_date_x9z2a` ✓

Readable AND unique. Better for debugging.

SLUG GENERATION CODE

```python
class SlugWithRandomSuffix:
"""Mixin for slug generation."""

def slug_with_random_suffix(self,
max_length: int = 50) -> str:
"""
"My Field" -> "my_field_x9z2a"
"""

CHARS = string.ascii_lowercase +
string.digits

name_part = slugify(self.label)
[:max_length - 6]
# Leave room for _xxxxx

random_part = "".join(
random.choice(CHARS)
for _ in range(5)
)

return f"{name_part}_{random_part}"
```

5-character suffix = 36^5 = 60 million possibilities — plenty for any tenant

Type Coercion: Everything is a String

The Challenge

HStore stores ALL values as strings. But users expect dates to behave like dates, numbers to sort correctly.

Problems This Creates

Sorting: ["1","10","2","9"] as strings
→ ["1","10","2","9"] ✗

Booleans: "True" vs "true" vs "1"

Solution: SerDe Layer

Serializer/Deserializer handles conversion between Python types and HStore strings.

```
# SerDe Mapping
SERDE_MAPPING = {
UdfType.TEXT: TextUdfSerDe(),
UdfType.INTEGER: IntegerUdfSerDe(),
UdfType.DATE: DateUdfSerDe(),
UdfType.BOOLEAN: BooleanUdfSerDe(),
UdfType.DECIMAL: DecimalUdfSerDe(),
UdfType.CONTACT: ContactUdfSerDe(),
}
```

DATEUDFSERDE EXAMPLE

```
class DateUdfSerDe:
"""Date: Python date <-> HStore string"""

def to_db(self, value: date) -> str:
"""Python -> HStore (for saving)"""
if value is None:
return ""
return value.isoformat() # "2023-01-15"

def from_db(self, raw: str) -> Optional[date]:
"""HStore -> Python (for reading)"""
if not raw:
return None
try:
return date.fromisoformat(raw)
except ValueError:
# Graceful degradation
logger.warning(f"Invalid:{raw}")
return None
```

ISO format for dates is crucial — it sorts correctly as strings! "2023-01-15" < "2023-02-01"

PART 3

Making It Work

LIVE DEMO

The Complete UDF Flow

Create a new UDF definition (**"Filing Date"** for Cases) in the admin panel

Open a Case edit form — see the new field appear **automatically**

Fill in the value and save

Verify data is stored and displayed correctly

The Complete Save/Load Flow

SAVING UDF VALUES

User submits form

↓

Form.clean() - Django validates

↓

For each UDF: validate, serialize

↓

instance.save() to HStore

LOADING UDF VALUES

Fetch instance from DB

↓

Build form dynamically

Deserialize each UDF value

↓

Render with Crispy Forms

Validation and type conversion happen in Python, not in the database

Frontend: Dynamic Form Factory

FACTORY: UDF → DJANGO FORM FIELD

```
def get_udf_form_field(udf: UDFDefinition,
initial_value=None):

common_kwargs = {
"label": udf.label,
"required": udf.is_required,
"initial": initial_value,
}

if udf.field_type == UdfType.TEXT:
return forms.CharField(
max_length=500, **common_kwargs)

elif udf.field_type == UdfType.DATE:
return forms.DateField(
widget=DatepickerWidget(),
**common_kwargs)

elif udf.field_type == UdfType.INTEGER:
return forms.IntegerField(
**common_kwargs)

# ... other types ...
```

INJECTING FIELDS AT RUNTIME

```
class CaseForm(forms.ModelForm):
class Meta:
model = Case
fields = ["title", "status"]

def __init__(self, *args, **kwargs):
self.user = kwargs.pop("user")
super().__init__(*args, **kwargs)

# Dynamically add UDF fields
udf_defs = UDFDefinition.objects \
.for_user(self.user) \
.filter(content_type=...)

for udf in udf_defs:
# Get existing value from HStore
raw = self.instance \
.user_defined_fields \
.get(udf.slug)
initial = udf.serde.from_db(raw)

self.fields[udf.slug] = \
get_udf_form_field(udf, initial)
```

Adding a new field type = adding one elif branch in the factory

Searching & Filtering UDF Values

DJANGO ORM SYNTAX

```
# Filter: priority = "high"
Case.objects.filter(
user_defined_fields__priority_x9z2a=
"high"
)

# Check if key exists
Case.objects.filter(
user_defined_fields__has_key=
"incident_date_p2q1m"
)

# Multiple conditions
Case.objects.filter(
user_defined_fields__priority_x9z2a=
"high",
user_defined_fields__incident_date__gte=
"2023-01-01"
)
```

GOTCHAS

String Comparison for Numbers

```
# WRONG: String comparison
...filter(quantity__gte="5")
# "10" < "5" as strings!
```

NULL vs Missing Key

```
# Key doesn't exist at all
{"priority": "high"}
# Key exists but empty
{"priority": "high", "date": ""}
```

Filter logic must handle both cases!

Good news: ISO date format sorts correctly as strings

`"2023-01-15" < "2023-02-01"` ✓

With GIN index, these queries are O(log n) instead of O(n) table scan

Advanced: Context Constraints

The Requirement

"I only want to see 'Bankruptcy Date' if the Case Type is 'Bankruptcy'."

Solution: UDFConstraint Model

- Stores a filter path (e.g., case_type)
- Stores allowed IDs whitelist
- Runtime check before rendering field

Result: Polymorphic Behavior

Without polymorphic database tables. Fields appear/disappear based on related data.

Why Whitelist? Performant (SQL IN clause), covers 95% of business cases. Expression engines are flexible but harder to query efficiently.

UDFCONSTRAINT MODEL

```
class UDFConstraint(models.Model):
"""Show field only for matching instances"""

field = models.ForeignKey(UDFDefinition,
on_delete=models.CASCADE)

related_model_lookup = models.CharField(
max_length=128) # e.g. "case_type"

ids_whitelist = ArrayField(
models.PositiveIntegerField())

def is_condition_met_by(self, obj):
"""Check if obj satisfies constraint"""
# Get FK id directly for efficiency
constrained_id = getattr(
obj,
f"{self.related_model_lookup}_id"
)
return constrained_id in
self.ids_whitelist
```

USAGE IN FORM

```
for udf in udf_definitions:
if udf.all_constraints_met_by(self.instance):
self.fields[udf.slug] = ...
```

Prevents "form clutter" — users hate seeing 50 irrelevant fields

Migration Strategy: Adding UDFs to Existing Systems

ROLLOUT STEPS

Add HStore Column
Zero downtime migration — just adds a nullable column

Create GIN Index
Use CONCURRENTLY to avoid table locks

Deploy Application Layer
UDFDefinition model, SerDe, form changes

Feature Flag Rollout
Enable for select clients first, monitor, then expand

Migrate Legacy Custom Columns (Optional)
Run data migration scripts to move existing data

DJANGO MIGRATION

```python
# Migration: Add HStore field

from django.contrib.postgres.fields \
import HStoreField

class Migration(migrations.Migration):

operations = [
# Step 1: Enable extension
HStoreExtension(),

# Step 2: Add the column
migrations.AddField(
model_name="case",
name="user_defined_fields",
field=HStoreField(
default=dict,
blank=True
),
),

# Step 3: Add GIN index
migrations.AddIndex(
model_name="case",
index=GinIndex(
fields=["user_defined_fields"]
),
),
]
```

This entire migration can be done with zero downtime on a live production system

PART 4

Wrap-Up

Key Takeaways

1. Schema-on-Read Works

HStore/JSONB gives flexibility without sacrificing query performance (with GIN indexes).

2. Metadata is Essential

UDFDefinition table provides the schema your application needs for validation, rendering, and querying.

3. SerDe Layer is Critical

Everything is a string in HStore. Your serializer/deserializer handles type safety at the application boundary.

4. ContentType = Reusability

Django's ContentType framework lets you write ONE generic system that works with any model.

Architecture at a Glance

This pattern scales to hundreds of custom fields with minimal performance impact



Thank You!

Questions & Discussion

The code patterns shown are battle-tested in production at Infino Legal