ChatGPT

Declarative UDF Framework for Himalytix ERP

We propose a modular framework that dynamically loads entity metadata (including user-defined fields or UDFs) from the database and uses it to generate models, forms, and validation logic. The **architecture** consists of: a *Metadata Layer* (calling a stored procedure to fetch schema), a *Model Generator* (producing dynamic Pydantic/Django classes), a *Form Builder* (rendering HTML/Jinja forms with validation attributes), a *Validation Dispatcher*, and a *Repository Layer* for persistence. The diagram below outlines the data flow:

- **Database** – Contains core tables (e.g. `chart_of_accounts` [1]) and UDF metadata tables (`udf_entity_types`, `udf_field_definitions`, etc. from schema [2] [3]).
- **Metadata Loader (Python)** – Calls a stored procedure (`sp_GetEntitySchema(entity_name)`) to retrieve schema (fields, types, labels, UDF rules) for the specified entity.
- **Schema Objects** – In-memory representations (e.g. Python dicts or dataclasses) of the metadata, including base fields and UDF definitions.
- **Model Generator** – Transforms schema objects into dynamic models (e.g. Pydantic or Django ORM models) with type hints, default values, and docstrings from labels.
- **Form Builder** – Takes a model/schema and renders an HTML or JSON form. Inputs are annotated with labels, placeholders, help text, and validation attributes (e.g. `required`, `pattern`). Custom widget hooks allow replacing any input (e.g. date pickers, selects).
- **Validation Dispatcher** – Uses the schema rules to validate incoming data. It performs built-in checks (type coercion, ranges, regex) and invokes any custom UDF-specific logic (e.g. Python callbacks or additional stored-proc checks). Errors are aggregated at field and form level.
- **Repository Layer (CRUD)** – Generic data-access classes (using Django ORM) handle creating/updating/deleting the main entity and saving UDF values. All changes are transactional and include the validation step.

Below we describe each module and data flow in detail.

1. Metadata Layer

Entity metadata (base fields + UDF definitions) is stored in SQL Server tables. For example, the `udf_field_definitions` table holds UDF info including field name, type, label, validation rules, etc. [2] [3]. We create a stored procedure `sp_GetEntitySchema(entity_name)` that joins these tables to return, for each field of the entity: - **Field name**, data type, label (display name), "required" flag. - **Validation metadata**: min/max, regex patterns, list-of-values (via `udf_field_options` table [4]), etc. - **UDF definitions**: type (string, number, date, reference, etc.), help text, default, and any custom logic indicators.

In Python, a **Schema Loader** module calls this stored procedure (e.g. via Django's DB connection). For example:

```
from django.db import connection
```

```
def load_entity_schema(entity_name: str, tenant_id: int = None):
    """Call sp_GetEntitySchema and return a schema dict."""
    with connection.cursor() as cursor:
        cursor.execute("EXEC sp_GetEntitySchema @EntityName=%s, @TenantId=%s",
[entity_name, tenant_id])
        columns = [col[0] for col in cursor.description]
        rows = cursor.fetchall()
    schema = [dict(zip(columns, row)) for row in rows]
    return schema
```

This returns a list of field metadata dicts. These can be deserialized into Python dataclasses or simple objects. **Caching:** To support hot-reload, the loader can cache schemas per-entity and invalidate when an admin changes UDFs (e.g. using a timestamp or a DB trigger).

2. Dynamic Model Generation

From the schema objects, we dynamically create Python models representing the entity. One approach is to use **Pydantic**'s `create_model`, which allows building a `BaseModel` subclass with fields at runtime. For example, given `schema_fields` list:

```
from pydantic import BaseModel, create_model, Field

def generate_pydantic_model(entity_name: str, schema_fields: List[Dict]):
    """Create a dynamic Pydantic model with fields from schema."""
    annotations = {}
    defaults = {}
    for fld in schema_fields:
        name = fld['FieldName']
        ftype = fld['DataType']  # e.g. 'string','int','date'
        # Map DB types to Python types
        if ftype in ('varchar','text'):    py_type = str
        elif ftype in ('int','bigint'):   py_type = int
        elif ftype in ('decimal','float'):py_type = float
        elif ftype == 'date':             py_type = str  # or datetime.date
        else:                             py_type = str
        default = fld.get('DefaultValue', None) or ...
        # Build annotation (type, default)
        annotations[name] = (py_type, default)
    DynamicModel = create_model(entity_name + "Model", **annotations,
__base__=BaseModel)
    # Add docstrings from labels
    for fld in schema_fields:
        setattr(DynamicModel.__fields__[fld['FieldName']], 'field_info',
fld['Label'])
    return DynamicModel
```

This model class has type hints, default values, and (optionally) Pydantic validators. For UDF-specific validation rules, we can attach `@validator` methods dynamically or override `validate()` to incorporate checks. Alternatively, for a Django-centric approach, one could dynamically create a `ModelForm` class or patch a Django `models.Model`, but using Pydantic keeps validation logic separate and reusable. The generated model's docstrings/field descriptions can come from the schema's `Label` or `HelpText`.

3. Form & Template Engine

A reusable **Form Builder** takes the model/schema and outputs a UI form. For server-rendered HTML (Bootstrap), we can loop over `schema_fields` and emit inputs. For example, in a Jinja2 template:

```
<form method="post">
{% for f in schema.fields %}
  <div class="form-group">
    <label for="{{ f.name }}">{{ f.label }}</label>
    <input id="{{ f.name }}" name="{{ f.name }}"
           type="{{ f.input_type }}"
           {% if f.required %} required {% endif %}
           placeholder="{{ f.placeholder }}"
           class="form-control"
           {% if f.pattern %} pattern="{{ f.pattern }}" {% endif %}/>
    {% if f.help_text %}
      <small class="form-text text-muted">{{ f.help_text }}</small>
    {% endif %}
  </div>
{% endfor %}
<button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Here, `schema.fields` is a list of metadata objects with attributes like `name`, `label`, `input_type` (e.g. text/date/select), `required`, `placeholder`, `pattern` (regex), `help_text`, etc., all coming from the database metadata. For select lists, `input_type='select'` and the builder would generate `<select>...options...</select>` using `udf_field_options` values. We also allow **widget overrides** via template hooks: e.g. if `f.widget == 'datepicker'`, render a date-picker widget.

In a **Django forms** context, one could use a form mixin. For example, the existing `DynamicFormMixin` in *Himalytix* (shown below) configures field properties at runtime based on a `field_config` dict. We can extend this idea: after instantiating a `forms.Form` or `ModelForm`, loop through our schema and use `form.add_field(name, field_instance, config)` to add each UDF field with attributes. For example, to add a new field:

```
from django import forms
```

```
class ChartAccountForm(DynamicFormMixin, forms.Form):
    # existing static fields
    account_code = forms.CharField(...)
    account_name = forms.CharField(...)
    def __init__(self, *args, field_config=None, **kwargs):
        super().__init__(*args, field_config=field_config, **kwargs)
        # Now add UDF fields based on schema
        for udf in schema_fields:
            field_type = udf['FieldType']
            if field_type == 'string':
                field = forms.CharField(required=udf['IsRequired'],
label=udf['DisplayName'])
            elif field_type == 'int':
                field = forms.IntegerField(...)
            # ... handle other types ...
            self.add_field(udf['FieldName'], field, config={
                'enabled': udf['IsActive'],
                'widget_attrs': {'placeholder': udf['Placeholder'] or ''},
                'css_classes': ['udf-field']
            })
```

This leverages the existing `DynamicFormMixin` (from [5]) which applies validators, CSS classes, and widget attributes based on a config dict [5] [6]. For example, one might extend it so that `form.add_field` not only adds the field but also updates `self.field_config` so that `configure_fields()` sets validators and attributes automatically.

4. Validation Utilities

Validation is driven entirely from the schema. After form submission, we first coerce types (e.g. convert strings to int/date). Then **built-in rules** are applied: e.g. if metadata says `min_value=0` or `regex='^\d{5}$'`, we enforce those. With a Pydantic model, most of these checks happen on instantiation (e.g. `DynamicModel(**data)` will raise errors if types or constraints fail). For **custom UDF logic**, we can provide hooks: for instance, store a Python callback or SQL query name in the metadata that is invoked during validation. Example:

```
errors = []
try:
    obj = DynamicModel(**user_input)
except ValidationError as ve:
    errors.extend(ve.errors())

# Custom UDF check: maybe a stored proc that must be called
if udf_custom_rule := udf['CustomRule']:
    is_valid, msg = call_custom_validator(udf_custom_rule, obj.dict())
```

```
    if not is_valid:
        errors.append({'loc': [udf['FieldName']], 'msg': msg})
```

All errors are then formatted per field. Form-level consistency checks (e.g. “end date after start date”) can also be encoded as UDFs or form cross-field validators. The end result is a structure of field-level and form-level messages to display to the user.

5. Persistence & CRUD

Once data passes validation, the **Repository Layer** saves it. We leverage Django’s ORM for core entities. For example, assume a `ChartAccount` model for base fields; we update it like any normal instance. UDF values, however, live in separate tables (`udf_field_values` and related tables from the schema). We implement generic UDF value handling, for example:

```
def save_entity_with_udfs(entity_name, entity_id, data: dict):
    # Save base entity fields via ORM
    if entity_name == 'ChartOfAccounts':
        acct = ChartOfAccounts.objects.get(pk=entity_id) if entity_id else
ChartOfAccounts()
        acct.account_code = data['account_code']
        acct.account_name = data['account_name']
        # ... set other fields ...
        acct.save()
        entity_id = acct.pk

    # Save UDF values (one row per UDF per entity)
    for udf in schema_fields:
        val = data.get(udf['FieldName'])
        if val is None and udf['FieldType']=='multi-select':
            # handle multi-select lists
            continue
        defaults = {'text_value': val if udf['FieldType']=='string' else None,
                    'number_value': val if udf['FieldType']=='int' else None,
                    # ... date_value, etc ...
                   }
        UdfFieldValue.objects.update_or_create(
            entity_id=entity_id, field_id=udf['FieldId'],
            defaults=defaults
        )
```

All of this is done in a transaction; if validation failed or any error is raised, the whole transaction rolls back. Delete operations similarly remove the base entity and cascade UDF values via foreign keys.

6. Extensibility & Configuration

Crucially, **no field lists or rules are hard-coded**. All configuration lives in the database tables [2] [3]. When an admin adds or edits a UDF (via the admin UI or a management command), the stored procedure output changes, and thus the next time forms/models are built they automatically include the new field. For performance, we cache schemas (in-memory or with Django cache) and invalidate them on changes. One can implement a simple version-check or timestamp (e.g. `udf_field_definitions.updated_at`) to know when to reload.

Because components are modular, one can plug them independently. For example, the model generator could be replaced by a code-gen script that writes Python files (using the same schema inputs), or the form builder could output JSON Schema for a React frontend instead of Jinja templates. The goal is **high cohesion, low coupling**: e.g. the validation logic relies only on the metadata, not on the form or model implementation details.

7. Documentation & Tests

We will auto-generate API docs from the dynamic models. For example, Pydantic models can be introspected for field docstrings (from the label/help), which tools like Sphinx or FastAPI's docs can use. Each module is documented with clear docstrings.

We write unit tests for every piece: - **Schema Loader tests**: Mock DB responses and ensure schema objects are parsed correctly. - **Model Generation tests**: Given a sample schema dict, check that the created Pydantic model has the right fields and defaults. - **Form Rendering tests**: Render a form and verify expected HTML tags and validation attributes appear for given schema. - **Validation tests**: Pass valid and invalid data to the validation dispatcher and assert correct error messages. - **Integration tests**: e.g., an end-to-end test that adds a UDF to the DB, then runs through the load-model-form-validate-save cycle for "Chart of Accounts" and verifies the UDF appears in forms and is persisted.

8. Data-Flow Sequence (Example)

1. **Admin defines a UDF**: e.g. "ChartOfAccounts has UDF 'ExternalCode' (string, 5-10 chars, required)". This updates tables like `udf_field_definitions`.
2. **User requests form**: The backend calls `load_entity_schema('ChartOfAccounts')` which returns metadata including `ExternalCode`.
3. The schema is converted to a Pydantic model and a form schema.
4. The HTML form is generated (with a new input for "External Code", marked required, with minlength=5, maxlength=10).
5. **User submits** the form. The input goes to the validation dispatcher, which uses the schema to check length and type.
6. If valid, the save handler creates/updates the `ChartOfAccounts` record and also writes the UDF value into `udf_field_values`.

This "zero-touch" flow means no code change was needed for the new UDF – the stored metadata drove everything.

9. Proof-of-Concept Code

Below are simplified code snippets illustrating core parts of the POC. They can be extended in your `Himalytix` repo (for example in a new Django app or within `components/core/`):

Metadata Loader (Python)

```
# metadata_loader.py
from django.db import connection

def get_entity_schema(entity_name: str, tenant_id: int = None):
    """
    Calls the stored procedure sp_GetEntitySchema and returns a list of field
metadata dicts.
    """
    with connection.cursor() as cursor:
        cursor.execute("EXEC sp_GetEntitySchema @EntityName=%s, @TenantId=%s",
[entity_name, tenant_id])
        cols = [col[0] for col in cursor.description]
        rows = cursor.fetchall()
    # Convert rows to list of dicts
    schema = [dict(zip(cols, row)) for row in rows]
    return schema

# Example usage:
# fields = get_entity_schema('ChartOfAccounts', tenant_id=42)
# # fields might include entries like {'FieldName': 'ExternalCode', 'DataType':
'varchar', 'IsRequired': True, 'MinLength': 5, ...}
```

Dynamic Pydantic Model Generator

```
# model_generator.py
from pydantic import BaseModel, create_model, Field

TYPE_MAP = {
    'varchar': (str, ...), 'text': (str, ...),
    'int': (int, ...), 'decimal': (float, ...),
    'date': (str, ...), 'boolean': (bool, ...),
    # add more mappings as needed
}

def create_dynamic_model(entity_name: str, fields: List[Dict]):
    """
    Given entity schema fields, create a Pydantic model class.
    """
```

```
    field_defs = {}
    for fld in fields:
        name = fld['FieldName']
        ftype = fld['DataType']
        ptype, default = TYPE_MAP.get(ftype, (str, ...))
        # Use None as default if not required
        default = None if not fld.get('IsRequired', False) else ...
        field_defs[name] = (ptype, default)
    DynamicModel = create_model(entity_name + "Model", **field_defs,
__base__=BaseModel)
    # Attach labels/help as metadata (optional)
    for fld in fields:
        if 'DisplayName' in fld:
            DynamicModel.__fields__[fld['FieldName']].field_info.description =
fld['DisplayName']
    return DynamicModel

# Example:
# ChartModel = create_dynamic_model('ChartOfAccounts', fields)
# instance = ChartModel(account_code="1000", account_name="Cash",
ExternalCode="ABC123")
```

Simple HTML Form Renderer (Jinja2)

```
{# templates/dynamic_form.html #}
<form method="post">
  {% for f in schema.fields %}
    <div class="form-group">
      <label for="{{ f.name }}">{{ f.label }}</label>
      {% if f.input_type == 'select' %}
        <select id="{{ f.name }}" name="{{ f.name }}" class="form-control"{% if
f.required %} required{% endif %}>
          {% for opt in f.options %}
            <option value="{{ opt.value }}">{{ opt.label }}</option>
          {% endfor %}
        </select>
      {% else %}
        <input id="{{ f.name }}" name="{{ f.name }}" type="{{ f.input_type }}"
               class="form-control"
               placeholder="{{ f.placeholder or '' }}"
               {% if f.required %} required{% endif %}
               {% if f.pattern %} pattern="{{ f.pattern }}"{% endif %}>
      {% endif %}
      {% if f.help_text %}
        <small class="form-text text-muted">{{ f.help_text }}</small>
      {% endif %}
```

```
    </div>
  {% endfor %}
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

In Python you would render this template with a context like:

```
schema = {
  'fields': [
    {'name': 'account_code', 'label': 'Code', 'input_type': 'text', 'required':
True, 'placeholder': 'e.g. 1001'},
    {'name': 'account_name', 'label': 'Name', 'input_type': 'text', 'required':
True},
    {'name': 'ExternalCode', 'label': 'External Code', 'input_type': 'text',
'required': False, 'placeholder': 'Alpha-numeric code'},
    # ... other fields ...
  ]
}
return render(request, 'dynamic_form.html', {'schema': schema})
```

Example: Chart of Accounts with Two UDFs

Suppose we add two UDFs for Chart of Accounts: 1. **ExternalCode** (string, required, max length 10)
2. **BudgetLimit** (decimal, optional).

We would insert these into `udf_field_definitions` via admin or SQL. Our loader will pick them up. The generated form will automatically include an **ExternalCode** text input with a `maxlength=10` (from metadata) and a **BudgetLimit** numeric input. Validation will enforce ExternalCode is non-empty and ≤10 characters, and that BudgetLimit (if provided) is a valid number.

Saving the form calls the repository logic above: it creates/updates the `ChartOfAccounts` record and also writes the UDF values into the `udf_field_values` table, linking them by `field_id`. The workflow requires no code changes when these UDFs are added – the schema proc and generator handle them dynamically.

Automated Tests (PyTest style)

```
def test_schema_loading(monkeypatch):
    # Mock the DB call to sp_GetEntitySchema
    monkeypatch.setattr(connection, 'cursor', mock_cursor_with([
        ('FieldName','DataType','IsRequired','DisplayName','Placeholder'),
        ('account_code','varchar',True,'Code',''),
        ('account_name','varchar',True,'Name',''),
        ('ExternalCode','varchar',False,'External Code','Enter code'),
    ]))
```

```
    fields = get_entity_schema('ChartOfAccounts', tenant_id=1)
    assert any(f['FieldName']=='ExternalCode' for f in fields)
    assert fields[0]['FieldName'] == 'account_code'

def test_model_generation():
    fields = [
        {'FieldName':'account_code','DataType':'varchar','IsRequired':True},
        {'FieldName':'ExternalCode','DataType':'varchar','IsRequired':False},
    ]
    Model = create_dynamic_model('ChartOfAccounts', fields)
    # Required field missing should error
    try:
        Model(ExternalCode='XYZ')
        assert False, "Validation should fail when required field missing"
    except ValidationError:
        pass
    # All fields present
    m = Model(account_code='A1', ExternalCode='XYZ')
    assert m.account_code == 'A1'

def test_form_rendering(client):
    schema = {
        'fields': [
            {'name':'ExternalCode','label':'External
Code','input_type':'text','required':False,'placeholder':'Code'},
            {'name':'BudgetLimit','label':'Budget
Limit','input_type':'text','required':False}
        ]
    }
    html = render_to_string('dynamic_form.html', {'schema': schema})
    assert 'name="ExternalCode"' in html and 'placeholder="Code"' in html
    assert 'name="BudgetLimit"' in html

def test_validation_rules():
    # Assume a validation util that uses Pydantic model
    Model = create_dynamic_model('Account', [

{'FieldName':'ExternalCode','DataType':'varchar','IsRequired':False,'MaxLength':
5},
    ])
    try:
        Model(ExternalCode='TOO_LONG_CODE')
        assert False
    except ValidationError as e:
        assert 'string too long' in str(e)
```

These examples illustrate how the pieces fit together. In your codebase (e.g. the `Himalytix` repo), you would integrate the metadata loader with Django settings (ensuring `DATABASES` is configured for MSSQL) and place generators/forms in appropriate modules (such as under `ERP/components` or a new app). The existing `DynamicFormMixin` in **Himalytix** (see code) [5] [6] can be extended to apply our UDF field configs to Django forms, ensuring attributes and validators from metadata are respected.

Summary: This design achieves **zero-touch UDF support**: adding or editing UDF records in the database immediately affects the behavior of forms and validation, without manual code changes. It separates concerns (metadata loading, model logic, form rendering, validation, persistence) into cohesive modules. Clear documentation and tests (as above) will help other developers onboard quickly and extend the framework for new field types or UI widgets.

Sources: The dynamic form mixin in the existing codebase [5] [6] and the database schema for UDF definitions [2] [3] inform this design. The Chart of Accounts table structure is documented in the schema [1], and our example UDFs build on that entity. The approach follows best practices for metadata-driven form frameworks.

---

[1] [2] [3] [4] erp_schema.sql
https://github.com/ritsnep/Himalytix/blob/8b005ac4db9b2b72619fdc3a1b701529d46af9b6/erp_schema.sql

[5] [6] forms.py
https://github.com/ritsnep/Himalytix/blob/8b005ac4db9b2b72619fdc3a1b701529d46af9b6/ERP/components/core/forms.py