# Django Form Builder

***Release 0.1***

**Università della Calabria**

**Sep 07, 2023**

# INSTALLATION


1 Requirements and Setup 3

2 Use single fields in your form 5

3 Fields methods and attributes 7

4 Create your own fields 9

5 Build dynamic forms 11

6 Create your DynamicFormClass and add static fields 15

7 Configure your project to use dynamic forms 17

8 Upload P7M and signed PDF files 21

9 Add/Remove Formset dynamically with javascript 23

A Django Framework application to build dynamic forms and define custom form fields types.

**Github:** https://github.com/UniversitaDellaCalabria/django-form-builder

**Features:**

- Forms definitions via JSON object;
- Save compiled form as JSON objects in model db and get its structure and contents with a simple model method call;
- Override form constructor in order to add static common fields;
- Create input fields using heritable classes, with customizable validation methods;
- Manage Django Formset fields, with form insertion and removal via javascript;
- Manage and verify digitally signed file fields (PDF and P7M) without a certification authority validation (TODO via third-party API).

CHAPTER

ONE

# REQUIREMENTS AND SETUP

Install ``django-form-builder`

Only `FileSignatureValidator` library is required as system dependency, it is needed to verify digitally signed attachments. See also requirements for python requirements.

```
pip install git+https://github.com/peppelinux/FileSignatureValidator.git
```

In `INSTALLED_APPS` include `django_form_builder` app.

```
INSTALLED_APPS = (
    # other apps
    'django_form_builder',
)
```

CHAPTER

TWO

# USE SINGLE FIELDS IN YOUR FORM

You can simply take single fields from this app and use them in you own form.
Just in your project include

```
from django_form_builder import dynamic_fields
```

and use every field as a normal form field

```
my_field = dynamic_fields.DynamicFieldClassName(params)
```

Every field has (or inherit) a `raise_error()` method that can be overrided to implement cleaning features and validation functions.

CHAPTER

THREE

# FIELDS METHODS AND ATTRIBUTES

`BaseCustomField` is the base class for every custom field.
This class defines two attributes and trhee foundamental methods that make fields work well.

**Attributes**

- `is_complex` (default *False*): if *True*, specifies that the field is composed by more elementar fields (like two DateFields);
- `is_formset` (default *False*): if *True*, specifies that the field is a Django Formset.

**Methods**

- `def define_value(self, custom_value=None, **kwargs)`: it integrates the field initialization with custom configuration parameters defined by user (e.g. choices of a SelectBox);
- `get_fields(self)`: if field *is_complex*, it returns a Python list of child fields. Else, it returns `[self]`;
- `def raise_error(self, name, cleaned_data, **kwargs)::` it integrates `clean()` method to have a customizable behaviour processing `cleaned_data`.

CHAPTER

FOUR

# CREATE YOUR OWN FIELDS

If you need to define your own fields inheriting an existing one, you can fastly create them by importing `dynamic_fields`

```
from django_form_builder import dynamic_fields
```

make an inheritance declaration

```
class MyCustomField(dynamic_fields.DynamicFieldClassName):
    # e.g. MyCustomField(BaseCustomField)
    ...
```

and override `get_fields()`, `define_value()` and `raise_error()` according to your needs.

CHAPTER

FIVE

# BUILD DYNAMIC FORMS

Now you can build your own form dynamically both in Django backend and frontend, just selecting the fields that you want, in total flexibility and easiness.

Every form can be saved in a configurable storage, in JSON format or simply defined in a Python Dictionary. Please see `django_dynamic_form.dynamic_fields` to see all the supported type.

```
from django_form_builder.forms import BaseDynamicForm
from collections import OrderedDict

constructor_dict = OrderedDict([('Telefono',  # field name
              ('CustomCharField',           # defines the FieldType
               {'label': 'Telefono',
                'required': True,
                'help_text': 'Fisso o Mobile',
                'pre_text': ''},            # a text to be rendered before the input␣
↪field
               '')),
             ('Credenziali attive dal',
              ('BaseDateField',
               {'label': 'Credenziali attive dal',
                'required': True,
                'help_text': 'Data di attivazione delle credenziali',
                'pre_text': ''},
               '')),
             ('al',
              ('BaseDateField',
               {'label': 'al',
                'required': True,
                'help_text': 'data di scadenza delle credenziali.',
                'pre_text': ''},
               '')),
             ('Descrizione Attività',
              ('TextAreaField',
               {'label': 'Descrizione Attività',
                'required': True,
                'help_text': "Descrizione dell'attività per la quale si richiedono le␣
↪credenziali",
                'pre_text': ''},
               '')),
             ('Richiede che le seguenti anagrafiche vengano attivate',
              ('CustomComplexTableField',  # a django fieldset
```

(continues on next page)

(continued from previous page)

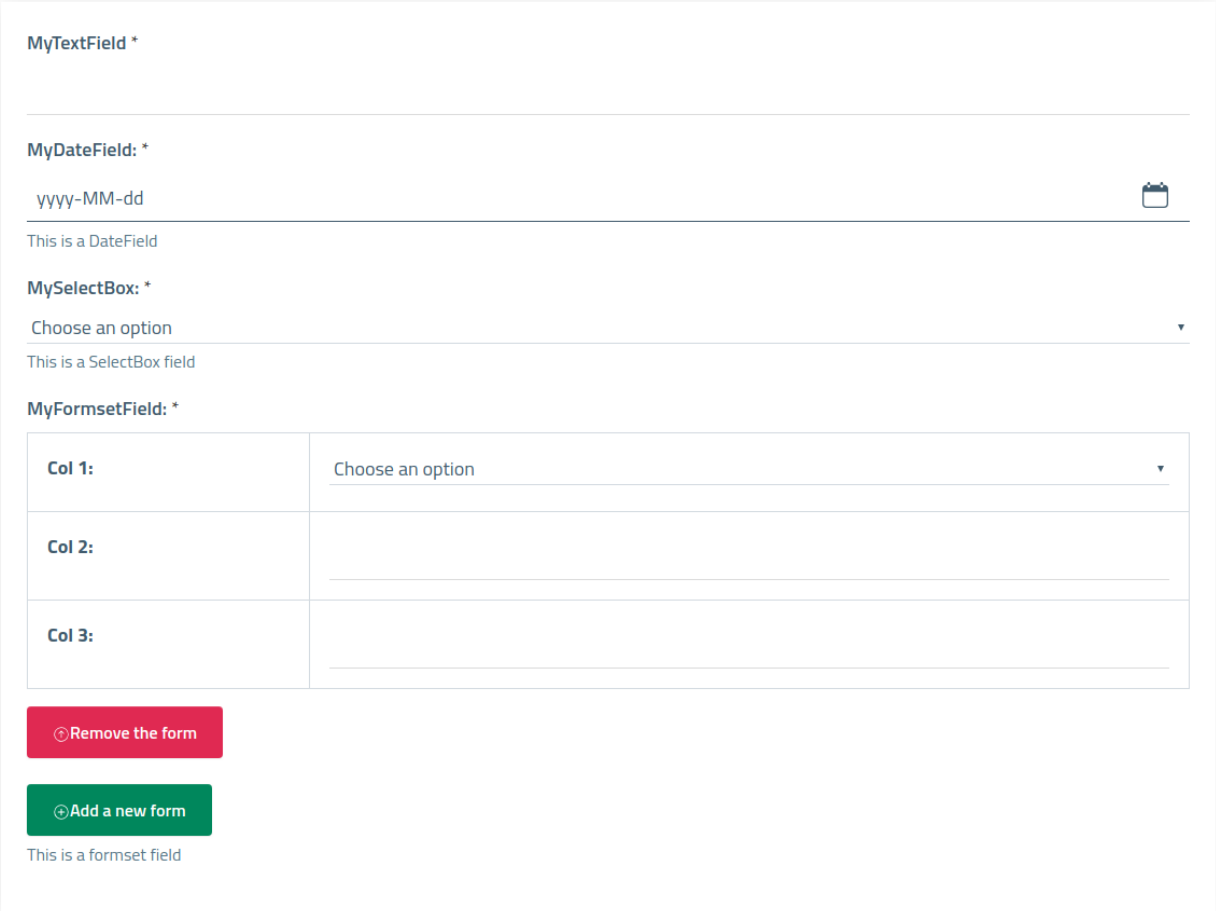
```
                {'label': 'Richiede che le seguenti anagrafiche vengano attivate',
                 'required': True,
                 'help_text': 'inserire almeno first_name, last_name e email',
                 'pre_text': ''},
                'first_name#last_name#place_of_birth#date_of_birth#codice_fiscale#email
↪#tel#valid_until'))])

form = BaseDynamicForm.get_form(#class_obj=YourCustomDynFormClass, # None by default,␣
↪then use BaseDynamicForm
                                constructor_dict=constructor_dict,
                                custom_params=None,
                                #data=data,   # if there's some data to load
                                #files=files, # if there's some file attachments (handled␣
↪separately)
                                remove_filefields=False,
                                remove_datafields=False)
```

*Example of Dynamic Form built via frontend:*

| Name | Field type | Value | Required | Help-text | Sort. | |
|---|---|---|---|---|---|---|
| MyTextField | CustomCharField | - | True | - | 0 | Edit Remove |
| MyDateField | BaseDateField | - | True | This is a DateField | 1 | Edit Remove |
| MySelectBox | CustomSelectBoxField | value1;value2;value3 | True | This is a SelectBox field | 2 | Edit Remove |
| MyFormsetField | FormsetField | col_1({'type':'CustomSelectBoxField', 'choices':'v1;v2;v3'})#col_2#col_3 | True | This is a formset field | 3 | Edit Remove |

*Preview of the builded form:*


MyTextField *
MyDateField: *
yyyy-MM-dd
This is a DateField
MySelectBox: *
Choose an option
This is a SelectBox field
MyFormsetField: *
Col 1:
Choose an option
Col 2:
Col 3:
Remove the form
Add a new form
This is a formset field

CHAPTER

SIX

# CREATE YOUR DYNAMICFORMCLASS AND ADD STATIC FIELDS

If you need some static field in your form, than you can define a new Form Class, inheriting *BaseDynamicForm*

```
from django_form_builder import dynamic_fields
from django_form_builder.forms import BaseDynamicForm

class MyDynamicForm(BaseDynamicForm):
    def __init__(self,
                 constructor_dict={},
                 custom_params={},
                 *args,
                 **kwargs):
       # Add a custom static field common to all dynamic forms
       self.fields = {}
       my_static_field = dynamic_fields.format_field_name(choice_field_name)
       my_static_field_data = {'required' : True,
                               'label': choice_field_label,
                               'help_text': choice_field_helptext}
       my_static_field = getattr(dynamic_fields,
                                 'CustomFieldClass')(**my_static_field_data)
       self.fields[my_static_field_id] = my_static_field

       # call super() constructor to build form
       super().__init__(# define it only if you
                        # define a custom field source,
                        # see "Create your own fields" paragraph.
                        # fields_source=dynamic_fields_integration,
                        initial_fields=self.fields,
                        constructor_dict=constructor_dict,
                        custom_params=custom_params,
                        *args, **kwargs)

    # if needed, override clean() method with your own params
    def clean(self, *args, **kwargs):
        cleaned_data = super().clean(own_param=own_value)
```

CHAPTER

SEVEN

# CONFIGURE YOUR PROJECT TO USE DYNAMIC FORMS

- **Step 1**

  Every Dynamic Form needs a table to store the list of fields that compose it.

  Also, it has to be strictly linked to a project model entity to be rendered (e.g. what kind of object wil the form map? A Book, a Car or something else!?).

  In your project's models, then, create a Model Class to store the list of fields, make it inherit class `DynamicFieldMap` and choose the ForeignKey that represents the form linked model entity.

```
from django_form_builder.dynamic_fields import get_fields_types
from django_form_builder.models import DynamicFieldMap

class MyFieldsListModel(DynamicFieldMap):
    """
    This class represents every single form field, each one linked to a unique␣
↪object
    """

    # if you want to integrate dynamic fields with your own,
    # define a new file that import all 'dynamic_fields' and defines others new and
    # then pass it as param to get_fields_types(class_name=my_custom_fields_file)

    my_entity = models.ForeignKey(MyEntityClass, on_delete=models.CASCADE)
    DynamicFieldMap._meta.get_field('field_type').choices = get_fields_types()
```

- **Step 2**

  Every submitted *dynamic form*, if valid, save its content as a JSON. Once we have our fields model (step 1), we have to define a Model Class to save our compiled form JSON attribute.

```
from django_form_builder.models import SavedFormContent

class MyModelClass(SavedFormContent):
    """
    This class contains the JSON with all submitted form details
    """
    ...
```

- **Step 3**

  In your views, use/override `get_form()` and `compiled_form()` methods to respectively build form structure from scratch (using your *step 1* model class) and rebuild and fill it simply by the JSON field.

```
from django_form_builder.models import DynamicFieldMap

...

# the class used as foreign key in 'Step 1'
class MyEntityClass(models.Model):

     ...

     def get_form(self,
                  data=None,
                  files=None,
                  remove_filefields=False,
                  remove_datafields=False,
                  **kwargs):
       """
       Returns the form (empty if data=None)
       if remove_filefields is not False, remove from form the passed FileFields
       if remove_datafields is True, remove all fields different from FileFields
       """
       # retrieve all the fields (the model class is in 'Step 1')
       form_fields_from_model = self.myfieldslistmodel.all().order_by('ordinamento')
       if not form_fields_from_model: return None
       # Static method of DynamicFieldMap that build the constructor dictionary
       constructor_dict = DynamicFieldMap.build_constructor_dict(form_fields_from_
↪model)

       # more params to pass with 'data'
       custom_params = {'extra_1': value_1,
                        'extra_2': value_2}
       # the form retrieved by calling get_form() static method
       form = DynamicFieldMap.get_form(# define it only if you
                                       # need your custom form:
                                       # class_obj=MyDynamicForm,
                                       constructor_dict=constructor_dict,
                                       custom_params=custom_params,
                                       data=data,
                                       files=files,
                                       remove_filefields=remove_filefields,
                                       remove_datafields=remove_datafields)

       return form
```

```
from django_form_builder.models import SavedFormContent

...

# the class used in 'Step 2'
class MyModelClass(SavedFormContent):

    ...
```

(continues on next page)

(continued from previous page)

```
    def compiled_form(self, files=None, remove_filefields=True):
        """
        Returns the builded and filled form
        Integrates django_form_builder.models.SavedFormContent.compiled_form
        SavedFormContent.compiled_form uses DynamicFieldMap.get_form() filled
        """
        # set get_form() source class (step 1)
        form_source = self.my_entity
        # set data source class (inherited from 'SavedFormContent')
        data_source = self.modulo_compilato

        form = SavedFormContent.compiled_form(data_source=data_source,
                                              files=files,
                                              remove_filefields=remove_filefields,
                                              form_source=form_source,
                                              **other_extra_params)

        return form
```

CHAPTER

EIGHT

# UPLOAD P7M AND SIGNED PDF FILES

Custom fields set provides a base class called `CustomSignedFileField` that via *FileSignatureValidator* library checks if an upload attachment is digitally signed.

Also, with `get_cleaned_signature_params()` method, it returns the sign details

- Signature Validation
- Signing Time
- Signer full Distinguished Name

P7M file fields are built by `CustomSignedP7MField(CustomSignedFileField)` class.
Signed PDF file fields are built by `CustomSignedPdfField(CustomSignedFileField)` class.

CHAPTER

NINE

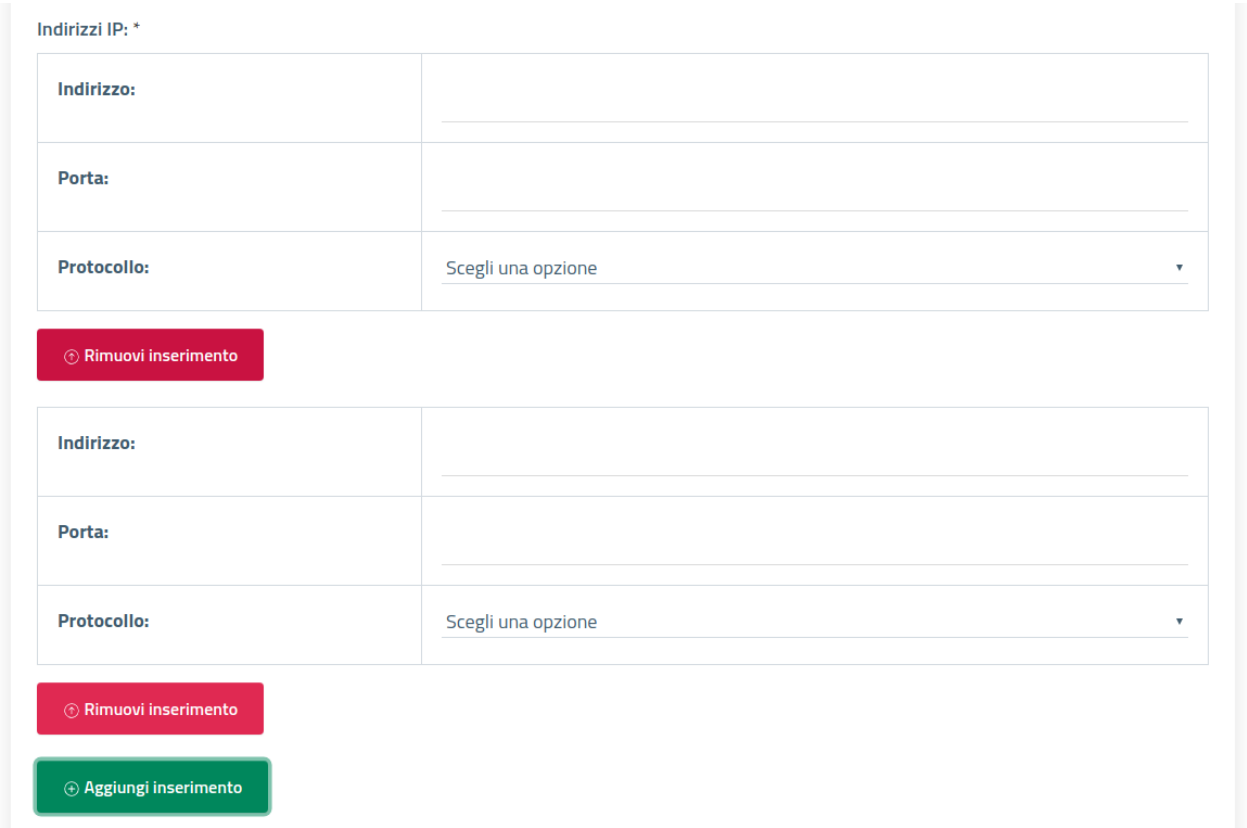
# ADD/REMOVE FORMSET DYNAMICALLY WITH JAVASCRIPT

Django Form Builder provides a particular type of field, `CustomComplexTableField`, that allows user to easily insert Django Formset Fields in his form.

The built-in javascript enables form inserting and removing via frontend, simply using the relative buttons!

To build a formset just define the `CustomComplexTableField` attribute *valore* setting columuns. Divide each one using # char and, for every column, define the field type with a dictionary, like in the example

```
column1({'type':'CustomSelectBoxField', 'choices': 'value1;value2;value3',})#column2({
↪'type':'CustomRadioBoxField', 'choices': 'value1;value2',})#column3#column4({'type':
↪'BaseDateField',})
```

Column with no params dict generate `CustomCharField` by default.

---