A PROJECT REPORT
ON

WORKING WITH DJANGO FORMS

Submitted to
SSN College of Engineering

BY

SUBALAKSHMI SHANTHOSI S 186001008

UNDER THE GUIDANCE OF
DR. R.S MILTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SSN COLLEGE OF ENGINEERING

Rajiv Gandhi Salai (OMR),Kalavakkam – 603 110
MAY-2019

Sri Sivasubramaniya Nadar College of Engineering(SSNCE)
(An Autonomous Institution, Affiliated to Anna University, Chennai)

WORKING WITH DJANGO FORMS

A PROJECT REPORT
ON
WORKING WITH DJANGO FORMS

Submitted by

SUBALAKSHMI SHANTHOSI S 186001008

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

MAY-2019

Sri Sivasubramaniya Nadar College of Engineering
(An Autonomous Institution, Affiliated to Anna University, Chennai)

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University, Chennai)
Rajiv Gandhi Salai (OMR), Kalavakkam - 603 110

BONAFIDE CERTIFICATE

Certified that this project report titled, **..."WORKING WITH DJANGO FORMS"** is the bonafide work of **"SUBALAKSHMI SHANTHOSI S(186001008)...."** who carried out the project work under my guidance.

SIGNATURE: **SIGNATURE:**

HEAD OF THE DEPARTMENT **SUPERVISOR**

Submitted for end semester project examination held on .....

INTERNAL EXAMINER

Acknowledgements

I am profoundly grateful to **Dr. R.S Milton** for his expert guidance and continuous encouragement throughout to see that this project reaches its target since its commencement to its completion.

We would like to express deepest appreciation towards **Dr. Chitra Babu** , **HOD**, SSN College of Engineering, Head of Department of Computer Engineering whose invaluable guidance supported me in completion this project.

At last I must express my sincere heartfelt gratitude to all the staff members of Computer Engineering Department who helped me directly or indirectly during this course of work.

SUBALAKSHMI SHANTHOSI S (186001008)

ABSTRACT

Working with django for designing customisable forms.
We begin by designing a basic form with primitive fields with in-build validation suite.Followed by creation of form it is been circulated to all the intended audience for filling whose responses are collected and stored in backstore.

The user responses are presented to the creator of the form for viewing and filtering with specific requirements.

Django functionality of building the **Model, Template and View (MTV)** for ensuring separation of concern and enhanced useability in the development process. Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- Preparing and restructuring data to make it ready for rendering.
- Creating HTML forms for the data.
- Receiving and Processing submitted forms and data from the client.

Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a **Form** class describes a form and determines how it works and appears.

Keywords: Form Design, Django-Form,Django Form Builder

Contents

Chapter 1

Introduction

1.1 Working with Django Forms

1.1.1 Problem Statement

Modelling a form creation interface with basic form fields like - Integer,Character and Pick-List Fields for manipulation.

Separation of tight relationship with the model and that of a trivial form POST method by introducting customisation like :

1. Building Form with json fields.
2. Python code for data population in Models instead of manual entry.

1.1.2 Requirements - Django Forms:

The specific user and system requirements for building form creation interface are listed below :

1. Creating Django forms which returns **structured data** on submission.
2. End-to-end working of form submission flow.
3. Customisation of form fields for enhanced useability.
4. Including in-place form validation effortlessly.
5. Customisation of form interface to handle JSON data.
6. Population script to automate entries to the data store.

Chapter 2

Introduction to HTML and CSS

2.1 HTML : Hyper Text Markup Language

2.1.1 Introduction to HTML

HTML is a **markup language** which tells how the computer should render a web page.
The documents themselves are plain text files with special "tags" or codes that a web browser uses to interpret and display information on your computer screen.

- Introduction
  1. An HTML file is a text file containing small markup tags.
  2. The markup tags tell the Web browser how to display the page.
  3. An HTML file must have an **htm** or **html** file extension.
- HTML Tags
  1. HTML Tags are used to markup HTML elements.
  2. HTML Tags are surrounded by < and > braces.
  3. HTML Tags are either:
     - Paired tags : Tags with opening and closing tags.
     - Closed tags : Tags without closing tag.

Tag	Description	Example
h1, . . . , h6	Header tag	h1 to h6
p	paragraph element	Hi ...
span	No line change after span	Hi...Bye.
div	Make division between contents	content
a	Hyperlink in documents	<a href ="<url>">description
center	Move content to center	Centered content
br	Line break	 or
hr	Horizontal line	<hr>or
pre	Preserve formatting	<pre>Preserved content</pre>
table	Include a table	<table><tr><td>TR One</td></tr></table>

Table 2.1: HTML basic tags

- **HTML 5**
  HTML 5 is backward compactible
  Features in HTML 5:

  1. **New semantic elements** to allow us to define more parts of our markup unambiguously and semantically rather than using lots of classes and IDs.
  2. **New elements and APIs** for adding video, audio, scriptable graphics, and other rich application type content to our sites.
  3. New features for standardising functionality that we already built in bespoke, hacky ways. **Server sent updates and form validation** spring to mind immediately.
  4. Customised handling of **markup errors**.
  5. Support for **Offline web pages**.

- **HTML 5 Design principles**:

  1. Ensuring support for existing content: Process existing HTML documents as HTML5 documents.
  2. Degrading new features gracefully in older browsers: HTML5 should degrade gracefully in old or less capable agents.
  3. Not reinventing the wheel : "If it ain't broke, don't fix it."
  4. Paving the cow paths: Work with already build path. Embrace and adopt the de facto standards in the official specs.
  5. Evolution, not revolution: Evolve existing system than to replace it with another new standards.

2.1.2 Introduction to CSS

CSS is used to enhance the look of the web page.
Types of CSS styling:

- Inline CSS :
  Style are defined inline to the individual tags:

```
<!-- css.html -->

<!DOCTYPE html>
<html>
<head>
<title>CSS Tutorial</title>
</head>

<body>

<h3 style="color:blue"> Heading 1 </h3>
<h3 style="color:red"> Heading 3 </h3>

</body>

</html>
```

- Embedded CSS: Styles are specified within the <style> tag in HTML document.

```
<!-- cssEmbedded.html -->
<!DOCTYPE html>
<html>
<head>
<title>CSS Tutorial</title>
<style type="text/css">
h3.h3_blue{ /*change color to blue*/
color: blue;
}

h3.h3_red{ /*change color to red*/
color:red;
}
</style>
</head>
<body>
<h3 class='h3_blue'> Heading 1 </h3>
<h3 class='h3_blue'> Heading 3 </h3>
<h3 class='h3_red'> Heading 1 </h3>
</body>
</html>
```

- External CSS: Style document in css extension is written separately which are imported into the HTML document.

```
/* asset/css/my_css.css */
h3.h3_blue{
color: blue;
}

h3.h3_red{
color:red;
}
```

```
<!-- CSS External -->
<head>
<title>CSS External file inclusion</title>
<link rel="stylesheet" type="text/css" href="asset/
  css/my_css.css">
</head>
```

- Basic CSS Selectors
  Three types of selectors in CSS :

  1. **Element:**Can be selected using it's name e.g. 'p', 'div' and 'h1' etc.
  2. **Class:** Can be selected using '.className' operator e.g. '.h3_blue'.
  3. **ID:**Can be selected using #idName e.g. '#my_para'.

```
/* asset/css/my_css.css */
/*element selection*/
h3 {
color: blue;
}
/*class selection*/
.c_head{
font-family: cursive;
color: orange;
}
```

```
/*id selection*/
#i_head{
font-variant: small-caps;
color: red;
}
```

```
<!-- css.html -->

<!DOCTYPE html>
<html>
<head>
<title>CSS Selectors</title>
<link rel="stylesheet" type="text/css" href="asset/css/my_css.css">
</head>
<body>
<h3>CSS Selectors</h3>
<p class='c_head'> Paragraph with class 'c_head' </p>
<p id='i_head'> Paragraph with id 'i_head' </p>
</body>
</html>
```

- Hierarchy: Hierarchy of the styling operations.
  - Priority level:
    1. ID(highest priority).
    2. Class selector.
    3. Element selector.
  - If two CSS has **same priority**, then CSS rule at the **last** will be applicable.

Selectors	Description
h1, p, span etc	Element selector
.className	Class selector
#idName	ID selector
*	Universal selector
h1.className	Selects heading one with class as Class Name
h1#className	Select h1 with id âĂŸidNameâĂŹ
p span	descendant selector (select span which is inside p)
p >span	child selector (âĂŸspanâĂŹ which is direct descendant of âĂŸpâĂŹ)
h1,h2,p	group selection (select h1, h2 and p)
span[my_id=m_span]	select âĂŸspanâĂŹ with attribute âĂŸmy_id=m_spanâĂŹ

Table 2.2: List of CSS selectors

- **CSS3**
  CSS3 aims at extending CSS2.1
  Features in CSS3:
  1. Rounded corners.
  2. Shadows.
  3. Gradients.
  4. Transitions or animations.
  5. Layouts - multi-columns.
  6. Flexible box or grid layouts.
- **CSS3 overview**:
  1. Borders:
     - border-color
     - border-image
     - border-radius
     - box-shadow
  2. Backgrounds:
     - background-origin and background-clip
     - background-size
     - multiple backgrounds

3. Colours:
   - HSL colors
   - HSLA colors
   - opacity
   - RGBA colors
4. Text Effects:
   - text-shadow
   - text-overflow
   - word-wrap
5. User Interface :
   - box-sizing
   - resize
   - outline
   - nav-top, nav-right, nav-bottom, nav-left
6. Basic box model :
   - overflow-x
   - overflow-y

- **CSS3 over CSS**:

1. **Compatibility**: CSS1 is not compatible with CSS3. CSS3 is backward compatibile with CSS1.
2. **Rounded corners and gradients**: Earlier versions of CSS required separate code to position and shape the figures or images.
   In CSS3

```
/* Round Border */
  .roundBorder{border-radius:10px;}"
  /* Gradient */
  .gradBG{background:liner-gradient(white,
    black);}
```

3. **Animation and Text Effects:** Animation in CSS using jQuery and jS scripts. No special text effects like shadow.CSS3 has text-shadow and break lines.CSS3 can also be used to add effects like hover can be set.

Chapter 3

Introduction to Django

3.1 Django Basics

3.1.1 Introduction : Django

Django (jang-goh) is a free and open source web application framework, written in Python.
A web framework is a set of components that helps you to develop websites faster and easier.
Every web designer aim at developing simple web components like User Management,Management panel for data filtering and viewing,forms etc.
The framework emphasis on reuseability and pluggability of components with less code , low coupling and rapid development.
The core Django Framework architecture can be seen as **MVC** architecture.
It consists of :

- Object-Relational mapper (**ORM**) that mediates between data models (defined as Python classes) and models.
- A relational database ("**Model**").
- A system for processing HTTP requests with a web templating system ("**View**").
- Regular-expression-based URL dispatcher ("**Controller**").

3.1.2 Design philosophies of Django

• Overall:
Fundamental philosophies Djang's developers have used in creating the framework. Its goal is to explain the past and guide the future.

1. **Loose Coupling and Tight Cohesion:**
   The various layers of the framework shouldn't "know" about each other unless absolutely necessary.

   For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn't care which template system a programmer uses.

2. **Less Code:**
   Django apps should use as little code as possible; they should lack boilerplate. Django should take full advantage of Python's dynamic capabilities, such as introspection.

3. **Quick development:**
   The point of a Web framework in the 21st century is to make the tedious aspects of Web development fast. Django should allow for incredibly quick Web development.

4. **Don't repeat yourself (DRY):**
   Redundancy is bad,normalisation is good.

5. **Consistency:**
   The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the "experience" of using Django).

Chapter 4

Models in Django

4.1 Introduction to Django Models

Models and Databases in Django:
A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

4.1.1 Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

Basic concepts:

- Each model is a Python class that subclasses **django.db.models.Model**.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API.

Using Models:
When you add new apps to INSTALLED_APPS, be sure to run **./manage.py migrate**,optionally making migrations for them first with **./manage.py makemigrations**.

4.1.2 Model Fields

Introduction

Each field in your model should be an instance of the appropriate Field class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. INTEGER, VARCHAR, TEXT).
- The default HTML widget to use when rendering a form field (e.g. <input type="text">, <select>).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Field Options

A set of common arguments available to all field types. All are optional. Most oftenly used field options are:

- **null** If True, Django will store empty values as NULL in the database. Default is False.
- **blank** If True, the field is allowed to be blank. Default is False.
- **choices** An iterable of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

```
YEAR_IN_SCHOOL_CHOICES = [
('FR', 'Freshman'),
('SO', 'Sophomore'),
('JR', 'Junior'),
('SR', 'Senior'),
('GR', 'Graduate'),
]
```

- **primary_key** primary_key If True, this field is the primary key for the model

4.1.3 Relationships

Types of Django models relationships:

- Many-to-one relationship.
- Many-to-Many relationship.

Many-to-One relationship

To define a many-to-one relationship, use **django.db.models.ForeignKey**. You use it just like any other Field type: by including it as a class attribute of your model. **ForeignKey** requires a positional argument: the class to which the model is related.

```
from django.db import models
class Manufacturer(models.Model):
# ...
pass

class Car(models.Model):
manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
```

Many-to-Many relationship

To define a many-to-many relationship, use **ManyToManyField**. You use it just like any other Field type: by including it as a class attribute of your model. **ManyToManyField** requires a positional argument: the class to which the model is related.

```
from django.db import models
class Topping(models.Model):
# ...
pass

class Pizza(models.Model):
# ...
toppings = models.ManyToManyField(Topping)
```

4.2 Model instance reference

4.2.1 Creating objects

To create a new instance of a model, just instantiate it like any other Python class: **class Model(**kwargs)**. Steps:

1. Add a class method on model class:

```
from django.db import models

class Book(models.Model):

  title = models.CharField(max_length=100)
  @classmethod
  def create(cls, title):
    book = cls(title=title)
    # do something with the book
    return book
  book = Book.create("Pride and Prejudice")
```

2. Add a method on a custom manager (usually preferred):

```
class BookManager(models.Manager):
  def create_book(self, title):
    book = self.create(title=title)
    # do something with the book
    return book
class Book(models.Model):
  title = models.CharField(max_length=100)
  objects = BookManager()
  book = Book.objects.create_book("Pride and Prejudice")
```

4.2.2 Validating objects

There are three steps involved in validating a model:

1. Validate the model fields - **Model.clean_fields()**.
2. Validate the model as a whole - **Model.clean()**.
3. Validate the field uniqueness - **Model.validate_unique()**.

4.2.3 Saving Model Object

To save an object back to the database, call save():

```
Model.save(force_insert=False, force_update=False,using=DEFAULT_DB_ALIAS,
    update_fields=None)
```

4.2.4 What happens when you save?

1. **Emit a pre-save signal**- The **pre_save** signal is sent, allowing any functions listening for that signal to do something.
2. **Preprocess the data**: Each field's pre_save() method is called to perform any automated data modification that's needed. For example, the date/time fields override pre_save() to implement **auto_now_add** and **auto_now**.
3. **Prepare the data for the database**: Each field's **get_db_prep_save()** method is asked to provide its current value in a data type that can be written to the database.
4. **Insert the data into the database**: The preprocessed, prepared data is composed into an **SQL statement** for insertion into the database.
5. **Emit a post-save signal**:The **post_save** signal is sent, allowing any functions listening for that signal to do something.

4.2.5 How Django knows to UPDATE vs. INSERT

- If the object's primary key attribute is set to a value that evaluates to **True** (i.e., a value other than None or the empty string), Django executes an **UPDATE**.
- If the object's primary key attribute is **not set** or if the UPDATE **didn't update** anything, Django executes an **INSERT**.

4.2.6 Use of Ordered Dict Type in Django

- Fields attribute mapping to appropriate type of Field classes.
- Fields attributes are not accessible by Form class object.
- Addition to field Dictonary on object creation.
- Addition of new field not possible directly in class.

4.3 Models in FormMason

In **models.py** include the classes with mentioned fields:

4.3.1 Writing Model classes

```
# Modelling rhe form database with a JSON file which is labelled by its title
class FormSchema(models.Model):
 title=models.CharField(max_length=100)
 schema=JSONField()
```

```
# Form Response – Responsible for retrieving the data and saving to backstore.
class FormResponse(models.Model):
 form=models.ForeignKey(FormSchema, on_delete=models.PROTECT)
 response=JSONField()

# Using custom_filter to obtain valid form entries.
 def is_form_valid(self,form):
   custom_form = FormSchema.objects.get(pk=self.kwargs["form_pk"])
   user_response = form.cleaned_data
   form_response = FormResponse(form=custom_form,response=user_response)
   form_response.save()
   return HttpResponseRedirect(reverse('home'))
```

4.3.2 Adding an extra field to a SampleForm instance

Using IPython shell mode by **./manage.py shell**:

```
In [9]: form=SampleForm()

In [10]: test_data={'name':'Subalakshmi Shanthosi S','age':24,'address':'Velache
    ...: ry ,Chennai','gender':'F'}

In [11]: form=SampleForm(test_data)

In [12]: from django import forms

In [13]: form.fields['country']=forms.CharField()

In [14]: form.is_valid()
Out[14]: False

In [15]: form.errors
Out[15]: {'country': ['This field is required.']}

In [16]:
```

4.3.3 Running Migration

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into.

There are several commands which you will use to interact with migrations and Django's handling of database schema:

1. **migrate**, which is responsible for applying and unapplying migrations.
2. **makemigrations**, which is responsible for creating new migrations based on the changes you have made to your models.
3. **sqlmigrate**, which displays the SQL statements for a migration.
4. **showmigrations**, which lists a project's migrations and their status.

```
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, main, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying main.0001_initial... OK
  Applying sessions.0001_initial... OK
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$
```

```
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$ python manage.py makemigr
ations main
System check identified some issues:

WARNINGS:
?: (2_0.W001) Your URL pattern '^admin/$' has a route that contains '(?P<', begi
ns with a '^', or ends with a '$'. This was likely an oversight when migrating t
o django.urls.path().
Migrations for 'main':
  main/migrations/0002_formresponse.py
    - Create model FormResponse
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$ python manage.py migrate
main
System check identified some issues:

WARNINGS:
?: (2_0.W001) Your URL pattern '^admin/$' has a route that contains '(?P<', begi
ns with a '^', or ends with a '$'. This was likely an oversight when migrating t
o django.urls.path().
Operations to perform:
  Apply all migrations: main
Running migrations:
  Applying main.0002_formresponse... OK
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$
```

4.3.4 Using created Model

```
In [1]: from main.models import FormSchema

In [2]: fs=FormSchema()

In [3]: fs.title='Custom Form One'

In [4]: fs.schema =  {'name': 'string', 'age': 'number', 'city': 'string',
   ...: 'country': 'string', 'time_lived_in_current_city': 'string'}

In [5]: fs.save()

In [6]: FormSchema.objects.get(pk=1).schema
Out[6]:
{'name': 'string',
 'age': 'number',
 'city': 'string',
 'country': 'string',
 'time_lived_in_current_city': 'string'}

In [7]:
```

Chapter 5

Forms in Django

5.1 Introduction

Django provides a rich framework to facilitate the creation of forms and the manipulation of form data.

- **The basics:**
  - Overview
  - Form API
  - Built-in fields
  - Built-in widgets
- **Advanced:**
  - Forms for models
  - Integrating media
  - Formsets
  - Customizing validation

5.1.1 Django's role in forms

Django handles three distinct parts of the work involved in forms:

- Preparing and Restructuring data to make it ready for rendering.
- Creating HTML forms for the data.
- Receiving and Processing submitted forms and data from the client.

5.1.2 Forms in Django

The Django Form class:

Django Model describes the logical **structure** of an object, it's **behavior**, and the way its parts are represented to us, a Form class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form <input> elements. (A **ModelForm** maps a model class's fields to HTML form <input> elements via a Form; this is what the Django admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A DateField and a FileField handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "**widget**" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Instantiating, processing, and rendering forms

While **rendering** a form:

- Get hold of it in the view (fetch it from the database, for example).
- Pass it to the template context.
- Expand it to HTML markup using template variables.

When we **instantiate** a form, we can opt to leave it empty or pre-populate it, for example with:

- Data from a saved model instance (as in the case of admin forms for editing).
- Data that we have collated from other sources.
- Data received from a previous HTML form submission.

Model Field	Form Field
AutoField	Not represented in form
BigAutoField	Not represented in form
BigIntegerField	IntegerField
BinaryField	CharField - True or false
BooleanField	BooleanField
CharField	CharField
DateField	DateField
ForeignKey	ModelChoiceField
ManyToManyField	ModelMultipleChoiceField
TextField	CharField

Table 5.1: Conversion of Model to Form Fields

5.2 Form Field Types

Each model field has a corresponding default form field. For example, a CharField on a model is represented as a CharField on a form. A model ManyToManyField is represented as a MultipleChoiceField.

5.2.1 Accessing the fields from the form

Using command **Form.fields**

```
>>> f.as_table().split('\n')[0]
'<tr><th>Name:</th><td><input name="name" type="text" value="instance" required
    ></td></tr>'
>>> f.fields['name'].label = "Username"
>>> f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value="instance"
    required ></td></tr>
```

5.2.2 Outputting forms as HTML

The second task of a Form object is to render itself as HTML. To do so, simply print it:

- **Form.as_p()**:
  as_p() renders the form as a series of **<p>** tags, with each <p> containing one field.
- **Form.as_ul()**: Similarly,Render with series of **<ul>** tags.
- **Form.as_table()**:Renders as HTML table.

5.3 Forms in FormMason

Form with inplace vaildation based on the field type and additional parameters such as max-length,min-length,required etc. specified on Form Fieldtype constructor.

```
# Form class which inherits django forms. Consists of four fields Name,Age,
    Address and Gender
from django import forms
class SampleForm(forms.Form):
  name = forms.CharField()
  age = forms.IntegerField()
  address = forms.CharField(required=False)
  gender = forms.ChoiceField(choices=(('M', 'Male'),('F','Female')))
```

Use of Ordered Dict Type in Django

Unlike the traditional Dictonary - the fields of Form are of type Ordered Dictionary. This is to **preserve** the order of insertion of field while writing forms.py.

5.3.1 Customised Form

New Dynamic Form(NewDynamicFormForm class) with the following fields as class members:

1. **form_pk** : CharField - required : False, HiddenInput.
2. **title** : CharField - required : True, Normal Input.
3. **schema** : CharField - required : True, Normal Input.
4. Method **clean_schema** : To trying loading json schema and returning schema.

```
import json
from django import forms
from django.db.models.functions import Lower,Upper

# Form fields - JSON data and title.
class NewDynamicFormForm(forms.Form):
 form_pk=forms.CharField(widget=forms.HiddenInput(),required=False)
 title = forms.CharField()
 schema = forms.CharField(widget=forms.Textarea())
# Method to load the JSON schema
def clean_schema(self):
 schema = self.cleaned_data["schema"]
 schema = json.loads(schema)
 return schema
```

- Alteration in views and template

```
#Create Edit Form View to load data from JSON into created model
class CreateEditFormView(FormView):
  form_class = NewDynamicFormForm
  template_name = "create_edit_form.html"
  def get_initial(self):
    if "form_pk" in self.kwargs:
      form = FormSchema.objects.get(pk=self.kwargs["form_pk"])
      initial = {
        "form_pk": form.pk,
        "title": form.title,
        "schema": json.dumps(form.schema)
      }
    else:
      initial = {}  # Ensure backward compactibility of old form layout
  return initial
# Get context data - Form context
def get_context_data(self, **kwargs):
  ctx = super(CreateEditFormView, self).get_context_data(**kwargs)
  if "form_pk" in self.kwargs:
    ctx["form_pk"] = self.kwargs["form_pk"]
  return ctx
# Save old or new form fields.
def form_valid(self, form):
  cleaned_data = form.cleaned_data
  if cleaned_data.get("form_pk"):
    old_form = FormSchema.objects.get(pk=cleaned_data["form_pk"])
    old_form.title = cleaned_data["title"]
    old_form.schema = cleaned_data["schema"]
    old_form.save()
  else:
    new_form = FormSchema(title=cleaned_data["title"],schema=cleaned_data["
        schema"])
    new_form.save()
    return HttpResponseRedirect(reverse("home"))
```

```
<!-- Changes to include link for edit form -- >
<! -- create_edit_form.html-->
{% extends "base.html" %}
{% block content %}
<h1>Create/Edit Form</h1>
{% if form_pk %}
<li>
<form action="{% url 'edit-form' form_pk=form_pk%}" method="post">
{% csrf_token %}
</li>
{% else %}
<li>
<form action="{% url 'create-form' %}" method="post">{% csrf_token %}
{% endif %}
</li>
{{ form.as_p }}
<li>
<input type="submit" value="Create Form" />
</li>
</form>
{% endblock %}
```

Chapter 6

Implementation

Form Builder implementation in Django using **Form** class.

6.1 Custom Form with predefined fields

Initial Form with four fields :

- Name - CharField.
- Age - IntegerField.
- Address - CharField.
- Gender - ChoiceField - Male or Female.

```
from django import forms
class SampleForm(forms.Form):
  name = forms.CharField()
  age = forms.IntegerField()
  address = forms.CharField(required=False)
  gender = forms.ChoiceField(choices=(('M', 'Male'), ('F','Female')))
```

```
subalakshmi@subalakshmi:~/WebApplicationDev/formMason$ ./manage.py shell
Python 3.6.5 |Anaconda, Inc.| (default, Apr 29 2018, 16:14:56)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from main.forms import SampleForm

In [2]: form = SampleForm()

In [3]:  form.fields
Out[3]:
OrderedDict([('name', <django.forms.fields.CharField at 0x7f2c180cb3c8>),
             ('age', <django.forms.fields.IntegerField at 0x7f2c180cb128>),
             ('address', <django.forms.fields.CharField at 0x7f2c180cbcc0>),
             ('gender', <django.forms.fields.ChoiceField at 0x7f2c180cbef0>)])

In [4]:
```

InfoBox 6.1: Ordered dictionary of fields

6.2 Adding an extra field to a SampleForm Instance

```
In [9]: form=SampleForm()

In [10]: test_data={'name':'Subalakshmi Shanthosi S','age':24,'address':'Velache
    ...: ry ,Chennai','gender':'F'}

In [11]: form=SampleForm(test_data)

In [12]: from django import forms

In [13]: form.fields['country']=forms.CharField()

In [14]: form.is_valid()
Out[14]: False

In [15]: form.errors
Out[15]: {'country': ['This field is required.']}

In [16]:
```

InfoBox 6.2: Adding new Form field in IShell.

6.3 Generating a form out of JSON

```
# main/views.py
class CustomFormView(FormView):
  template_name="custom_form.html"
  #success_url = reverse_lazy('custom_form')

def form_valid(self, form):
  custom_form = FormSchema.objects.get(pk=self.kwargs["form_pk"])
  user_response = form.cleaned_data
  form_response = FormResponse(form=custom_form, response=user_response)
  form_response.save()
  return HttpResponseRedirect(reverse('home'))

def get_form(self):

  # Form structure is predefined with formSchemaIns.schema with the below given
     json and have done formSchemaIns.save() to load the DB schema
  form_structure = FormSchema.objects.get(pk=1).schema
  #form_structure_json="""{"name":"string","age":"number","city":"string","
     country":"string","time_lived_in_current_city":"string"}"""
  #form_structure=json.loads(form_structure_json)
  custom_form=forms.Form(**self.get_form_kwargs())
  for key,value in form_structure.items():
  field_class=self.get_field_class_from_type(value)
  if field_class is not None:
    custom_form.fields[key]=field_class()
  else:
    raise TypeError("Invalid field type {}".format(value))

  return custom_form

def get_field_class_from_type(self,value_type):
  if value_type=="string":
    return forms.CharField
  elif value_type == "number":
    return forms.IntegerField
  else:
    return None

def get_context_data(self, **kwargs):
  ctx = super(CustomFormView, self).get_context_data(**kwargs)
  form_schema = FormSchema.objects.get(pk=self.kwargs["form_pk"])
  ctx["form_schema"] = form_schema
  return ctx
```

```
<!--  main/templates/custom_form.html-->
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8"/>
<title>Custom Form Demo</title>
</head>
<body>
<h1>Custom Form</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

```
from django.conf.urls import url
from main.views import CustomFormView
urlpatterns = [
url(r'^$', CustomFormView.as_view(), name='custom-form'),
]
```

6.4 A model for our JSON

Package used : **django-jsonfield**

- Changes in models:
- Form Title as CharField
- JSON schema as JSONField of django type.

```
# A model consisting of two fields namely : Schema in JSON format and form title
    .
from __future__ import unicode_literals
from django.db import models
from jsonfield import JSONField

class FormSchema(models.Model):
  title = models.CharField(max_length=100)
  schema = JSONField()
```

```
In [1]: from main.models import FormSchema

In [2]: fs=FormSchema()

In [3]: fs.title='Custom Form One'

In [4]: fs.schema =  {'name': 'string', 'age': 'number', 'city': 'string',
   ...: 'country': 'string', 'time_lived_in_current_city': 'string'}

In [5]: fs.save()

In [6]: FormSchema.objects.get(pk=1).schema
Out[6]:
{'name': 'string',
 'age': 'number',
 'city': 'string',
 'country': 'string',
 'time_lived_in_current_city': 'string'}

In [7]:
```

InfoBox 6.3: Saving changes.

6.5 Creating a better user interface

Better user interface by presenting the form_data in tabular format. Changes in **settings.py** to include new configuration of templates.

```
TEMPLATES = [
{
'BACKEND': 'django.template.backends.django.DjangoTemplates',
'DIRS': [
os.path.join(BASE_DIR, 'templates'),
],
'APP_DIRS': True,
'OPTIONS': {
'context_processors': [
  'django.template.context_processors.debug',
  'django.template.context_processors.request',
  'django.contrib.auth.context_processors.auth',
  'django.contrib.messages.context_processors.messages',
  ],
  },
},
]
```

Changes in **base.html**

```
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8"/>
<title>Form Mason</title>
</head>
<body>
  <a href="{% url 'home' %}">Home</a>
  {% block content %}
  {% endblock %}
</body>
</html>
```

Changes in **main/templates/custom_form.html**

```
{% extends "base.html" %}
{% block content %}
<h1>Custom Form</h1>
  <form action="" method="post">{% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Submit" />
  </form>
{% endblock %}
```

Changes in **main/views.py**

```
from django import forms
from django.views.generic import FormView
from django.views.generic import ListView
from main.models import FormSchema

class HomePageView(ListView):
  model = FormSchema
  template_name = "home.html"

class CustomFormView(FormView):
    template_name = "custom_form.html"

    def get_form(self):
      form_structure = FormSchema.objects.get(pk=self.kwargs["form_
    pk"]).schema
      custom_form =forms.Form(**self.get_form_kwargs())

      for key, value in form_structure.items():
      field_class = self.get_field_class_from_type(value)

      if field_class is not None:
        custom_form.fields[key] = field_class()
        else:
```

```
            raise TypeError("Invalid field type {}".format(value))
        return custom_form

    def get_field_class_from_type(self, value_type):
        if value_type == "string":
            return forms.CharField
        elif value_type == "number":
            return forms.IntegerField
        else:
            return None
```

URL configuration:

```
from django.conf.urls import url
from main.views import CustomFormView
from main.views import HomePageView
  urlpatterns = [
  url(r'^$', HomePageView.as_view(), name='home'),
  url(r'^form/(?P<form_pk>\d+)/$', CustomFormView.as_view(),name='custom-form'),
]
```

6.6 Saving responses of Django Forms

Changes to view the responses saved in backend in table:

Changes in **main/views.py**

```
class FormResponsesListView(TemplateView):
  def get_context_data(self, **kwargs):
    ctx = super(FormResponsesListView, self).get_context_data(**kwargs)
    form = self.get_form()
    schema = form.schema
    form_fields = schema.keys()
    ctx["headers"] = form_fields
    ctx["form"] = form
    responses = self.get_queryset()
    responses_list = list()
    for response in responses:
      response_values = list()
      response_data = response.response
      for field_name in form_fields:
        if field_name in response_data:
          response_values.append(response_data[field_name])
        else:
          response_values.append('')
      responses_list.append(response_values)
    ctx["object_list"] = responses_list
    return ctx
```

```
def get_queryset(self):
  form = self.get_form()
  return FormResponse.objects.filter(form=form)

def get_form(self):
  return FormSchema.objects.get(pk=self.kwargs["form_pk"])
```

Changes in form_response template to construct table:

```
{% extends "base.html" %}
{% block content %}
<h1>Responses for {{ form.title }}</h1>
  {% if object_list %}
    <ul>
    {% for response in object_list %}
      <li>{{ response.response }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endblock %}
```

Changes in URL configurations:

```
from main.views import FormResponsesListView

urlpatterns = [ ... ,

  url(r'^form/(?P<form_pk>\d+)/responses/$', FormResponsesListView.as_view(),
      name='form-responses'),
]
```

Changes made in home template to include link of newly included view function.

```
{% extends "base.html" %}
{% block content %}
  <h1>Responses for {{ form.title }}</h1>
  {% if object_list %}
  <table border="1px">
    <tr>
    {% for field_name in headers %}
      <th>{{ field_name }}</th>
    {% endfor %}
    </tr>
    {% for response in object_list %}
    <tr>
```

```
    {% for field_value in response %}
    <td>{{ field_value }}</td>
    {% endfor %}
    </tr>
    {% endfor %}
  </table>
  {% endif %}
{% endblock %}
```

Home

Responses for Custom Form One

name	age	city	country	time_lived_in_current_city
Subalakshmi S	24	Chennai	India	Today
Aruna S	27	Dubai City	Dubai	Last Year
Subalakshmi S	24	Chennai	India	Today
Subalakshmi S	27	Chennai	India	Today
Saradha	24	Pondy	India	Two Years
Dharini	24	Tirunelveli	India	Six months

InfoBox 6.4: Tabular view of form responses.

6.7 Designing a form creation interface

Changes in **main/forms.py** to load a JSON which is part of model field got from validation function **self.cleaned_data**

```
import json
from django import forms
class NewDynamicFormForm(forms.Form):
  form_pk = forms.CharField(widget=forms.HiddenInput(),required=False)
  title = forms.CharField()
  schema = forms.CharField(widget=forms.Textarea())

  def clean_schema(self):
    schema = self.cleaned_data["schema"]
    try:
      schema = json.loads(schema)
    except:
      raise forms.ValidationError("Invalid JSON. Please submit valid JSON for
          the schema")
    return schema
```

CreateEditFormView - class with following methods and instance variables

- form_class- NewDynamicFormForm
- template_name- create_edit_form.html
- get_initial - get_initial Schema if FormSchema object exists else use new schema with title,form_pk and schema in JSON format , returns initial schema which was formed.
- get_context_data - same purpose to use inherited reference to CreateEditFormView
- form_valid does the following functionality:
  - Get form cleaned_data
  - Check if Any object already found
  - Get Schema old with corresponding pk
  - Set title
  - Set Schema
  - Save form schema

Flow Alternative

- New Form Flow - Create new FormSchema
- Save Schema
- Return to home.html

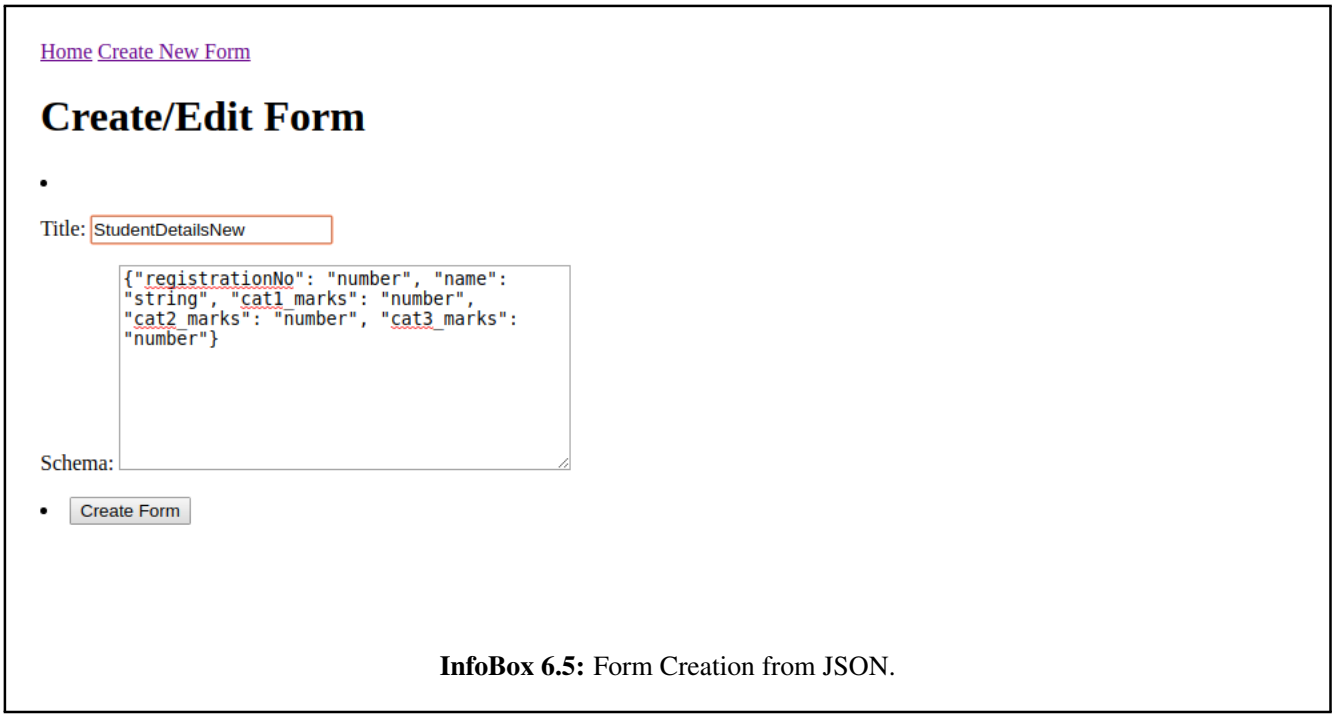
Changes in **create_edit_form.html**

Include a link to create-edit panel

```
{% extends "base.html" %}
{% block content %}
<h1>Create/Edit Form</h1>
{% if form_pk %}
<form action="{% url 'edit-form' form_pk=form_pk%}" method="post">{%
csrf_token %}
{% else %}
<form action="{% url 'create-form' %}" method="post">{% csrf_token %}
{% endif %}
{{ form.as_p }}
<input type="submit" value="Create Form" />
</form>
{% endblock %}
```

```
class CreateEditFormView(FormView):
 form_class = NewDynamicFormForm
 template_name = "create_edit_form.html"

 def get_initial(self):
   if "form_pk" in self.kwargs:
     form = FormSchema.objects.get(pk=self.kwargs["form_pk"])
     initial = {
       "form_pk": form.pk,
       "title": form.title,
       "schema": json.dumps(form.schema)
     }
   else:
      initial = {}

   return initial

def get_context_data(self, **kwargs):
   ctx = super(CreateEditFormView,self).get_context_data(**kwargs)
   if "form_pk" in self.kwargs:
     ctx["form_pk"] = self.kwargs["form_pk"]
   return ctx

def form_valid(self, form):
   cleaned_data = form.cleaned_data
   if cleaned_data.get("form_pk"):
     old_form = FormSchema.objects.get(pk=cleaned_data["form_pk"])
     old_form.title = cleaned_data["title"]
     old_form.schema = cleaned_data["schema"]
     old_form.save()
   else:
     new_form = FormSchema(title=cleaned_data["title"],schema=cleaned_data["
         schema"])
     new_form.save()
     return HttpResponseRedirect(reverse("home"))
```

```
 url(r'^$', HomePageView.as_view(), name='home'),
 url(r'^form/(?P<form_pk>\d+)/$', CustomFormView.as_view(),
 name='custom-form'),
 url(r'^form/(?P<form_pk>\d+)/responses/$', FormResponsesListView.
 as_view(), name='form-responses'),
 url(r'form/new/$', CreateEditFormView.as_view(), name='createform'),
 url(r'form/(?P<form_pk>\d+)/edit/$', CreateEditFormView.as_view(),
 name='edit-form'),
 ]
```



InfoBox 6.5: Form Creation from JSON.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

Detailed overview of form design and rendering in Django.
Realising the objective of user-friendly forms for better reach of the application.
Few advanced topics like FormsAPI and Django application deployment are left for future exploration.

7.2 Future Scope

There are many open-ended improvements or customisation to be introduced for enhancing the usability of form creation interface of which one requirement of creating form from JSON schema was experimented with a simple GUI.
Few open tasks as improvements in future are listed below:

- Enhancing the form creation interface by introducing drag and drop field palette.
- Ehhancing the form creation interface by cripsy-forms : to control the rendering behavior of your Django forms in a very elegant and DRY way by a **{%crispy%}** or **| crispy** filter.
- Using Django-tables to render the tabular form of responses instead of redesigning the whole template.
- Writing functional tests for better code coverage and understandability.

References

[1] HTML Guide.

[2] HTML Basics.

[3] HTML5 and CSS3 Basics.

[4] CSS3-Documentation

[5] Django documentation.

[6] Django Design philosophies.

[7] Django documentation.

[8] Django Project Blueprint.

[9] Django Project Blueprints.

[10] Django Forms.

[11] Bootstrap 4 Forms in Django.

[12] Bootstrap 4 Forms in Django using crispy forms.