2 Using a Web Application Framework

Processing Requests, Templates, Model-View-Controller

Web Application Framework

Server-side Frameworks

- defines a set of rules and provides an architecture
  - to create web pages, landings and forms
  - to provide a larger set of functionality compared to client-side pages
  - handle
    - HTTP-requests
    - URL mapping
    - database access
    - HTML generation
  - improve security

Web Application Framework

Python's Frameworks

Parameter	Django	Flask
Type of Framework	full-stack web framework	lightweight framework
Data Model	object-oriented approach that enables object-relational mapping to different relational databases	simple modular approach
Project Layout	multiple page applications	single-page application
Flexibility	less due to built-in tools	extensible
Template Engine	built-in model view template	Ninja2 template design
Routing and Views	supports the mapping of URL to views	mapping of URL to class-based view

III.2

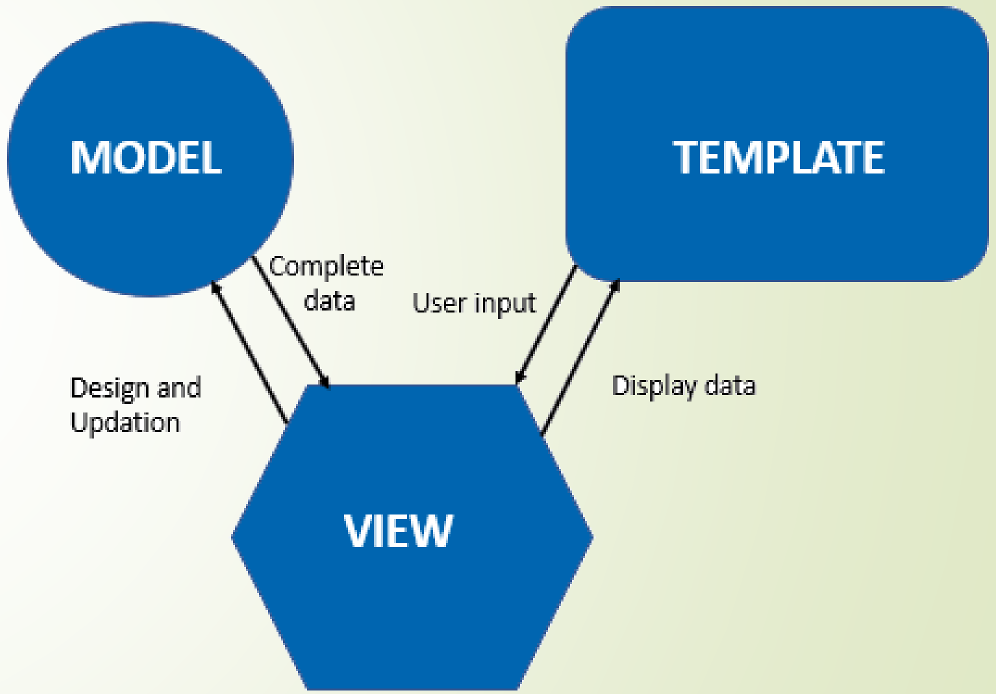
Django – Concepts

Model, View, Template (MVT)

- Model
  does the linking to the database
  each model gets mapped to
  a single table in the database

- Template
  handles the UI and architecture
  part of an application

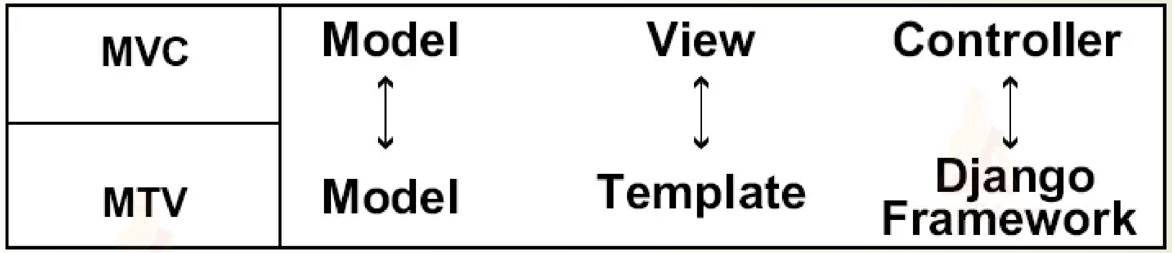
- View
  does the logical part of the application
  and interacts with the Model to get
  the data and in turn modifies the template accordingly



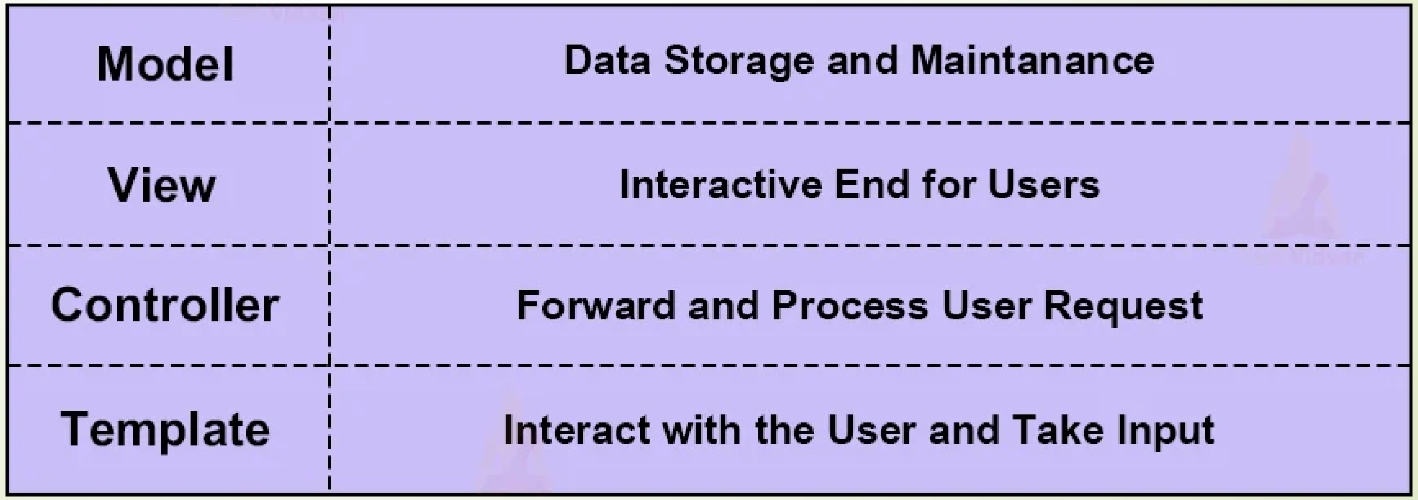
Django – Concepts

Model, View, Template (MTV)

- MVC versus MTV



MVC	Model	View	Controller
	↕	↕	↕
MTV	Model	Template	Django Framework



Model	Data Storage and Maintanance
View	Interactive End for Users
Controller	Forward and Process User Request
Template	Interact with the User and Take Input

Django – Project Structure

Project Structure

- framework distinguishes
  - project
    - top-level unit of organisation
    - contains elements used in the whole web application
  - app
    - lower-level unit of organisation

Django – Project Structure

Preparation

- installing Django in VSC
  - pip install Django
  - from the Command Palette (Ctrl+Shift+P), select "Python: Start REPL command" to open a REPL terminal for the currently selected Python interpreter
    - import django
    - print(django.get_version())
- setting up a workspace

III.2

Django – Project Structure

Project Structure

- create the website

```
django-admin.exe startproject myWebproject
```

➢ myWebproject/	top-level project folder
➢ myWebproject/myWebproject/	lower-level folder that represents the site
➢ manage.py	serves as the command center of the project

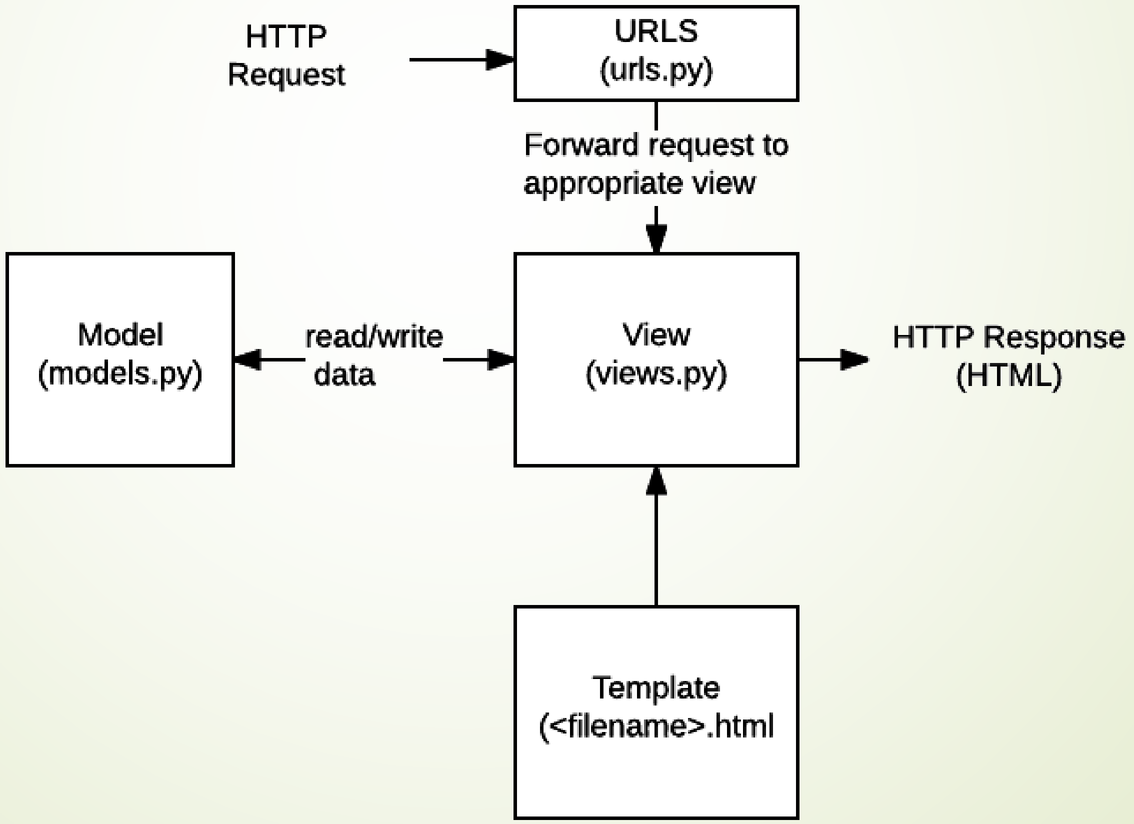
- prepare for debugging
  - ➢ create a launch.json
- create an app

```
python manage.py startapp app1
```

➢ myWebproject/app1	web application

Django – Project Structure

Project Structure – Web application

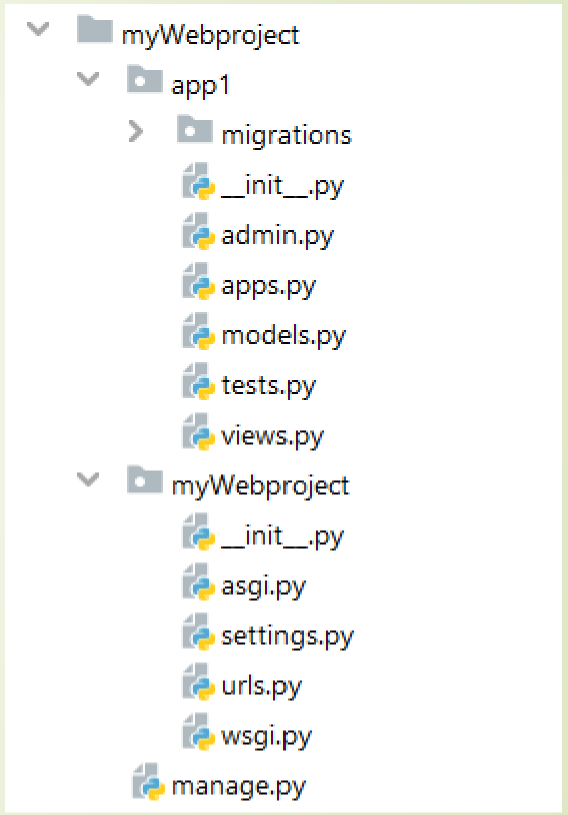


III.2

Django – Project Structure

Project Structure – Web site

- settings.py
  contains all the website settings, including registering any applications, the location of static files, database configuration details, etc.
- urls.py
  defines the site URL-to-view mappings
  while this could contain all the URL mapping code,
  it is more common to delegate some of the mappings
  to particular applications
- wsgi.py / asgi.py
  is used to help your Django application communicate with the webserver



```
myWebproject
    app1
        migrations
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        views.py
    myWebproject
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
    manage.py
```

III.2

Django – Project Structure

Project Structure – Web application

- __init__.py
  to declare a folder as a package, which allows Django to use code from different apps to compose the overall functionality of your web application
- models.py
  declare the app's models in this file, which allows Django to interface with the database of the web application
- views.py
  contains most of the code logic of the app

III.2

Django – Project Structure

Project Setup

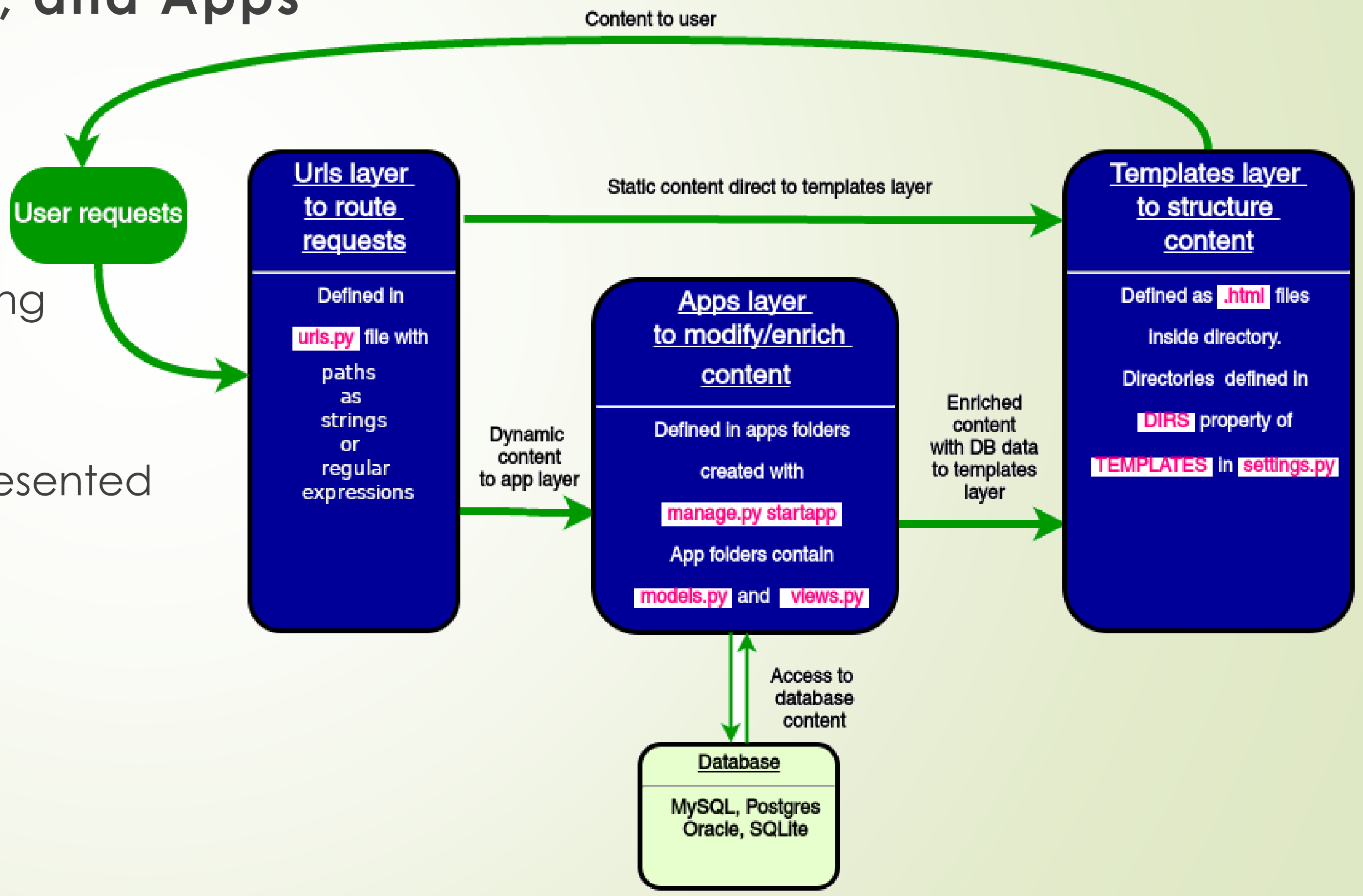
- run the initial migrations to set up the database schema
  - python manage.py migrate
- start the development server
  - python manage.py runserver

III.2

Django – Layers

URLs, Views&Templates, and Apps

- pipelines for static and for dynamic content
  - URLs-layer
    route requests by mapping URLs to views
  - Apps-layer
    computes data to be presented
  - Templates-layer
    generates HTML-pages


Content to user
User requests
Urls layer to route requests
Defined in
urls.py file with
paths as strings or regular expressions
Static content direct to templates layer
Templates layer to structure content
Defined as .html files
inside directory.
Directories defined in
DIRS property of
TEMPLATES in settings.py
Dynamic content to app layer
Apps layer to modify/enrich content
Defined in apps folders
created with
manage.py startapp
App folders contain
models.py and views.py
Enriched content with DB data to templates layer
Access to database content
Database
MySQL, Postgres Oracle, SQLite

Django – Concepts

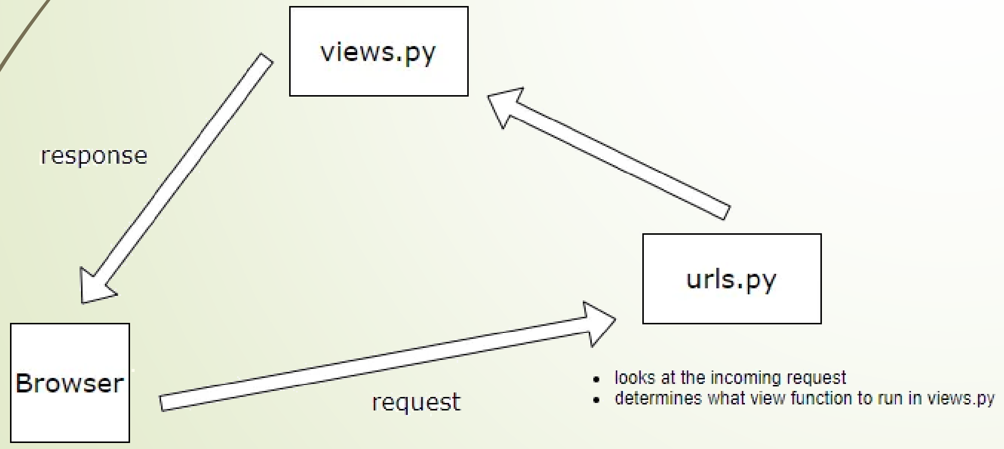
Project Activities

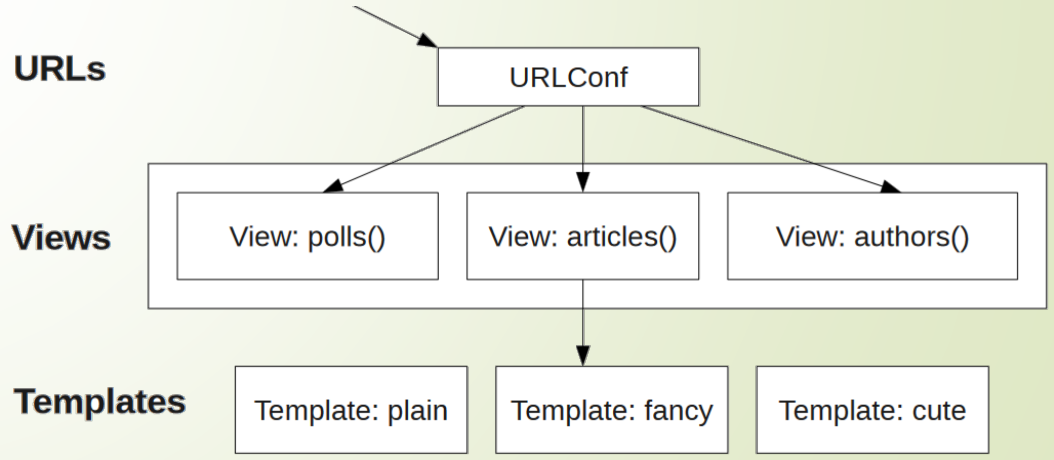
- register and configure an applications to include them in the project
  ↳ settings.py
- structure the project into different modules/views and define URL-patterns for these views
- create mappers to associate the URL patterns with specific views
- create views to retrieve specific data in response to different requests, and templates to render the data as HTML to be displayed in the browser

Django – URL Layer

URLs

- Django's concept
  URL patterns map to Views, Views may use templates, Templates contain HTML
- urls.py specifies how URL's get mapped to views


views.py
response
urls.py
Browser
request
looks at the incoming request
determines what view function to run in views.py



Django – URL Layer

URL-Dispatcher

- Django determines the root URLconf module to use, loads that Python module and looks for the variable `urlpatterns`. This should be a sequence of `django.urls.path()` and/or `django.urls.re_path()` instances
- Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL, matching against `path_info`
- Once one of the URL patterns matches, Django imports and calls the given view, which is a Python function (or a class-based view). The view gets passed the following arguments:
  - `HttpRequest`
  - URL-patterns matched to regular expressions

Django – URL Layer

URL-Dispatcher

- example

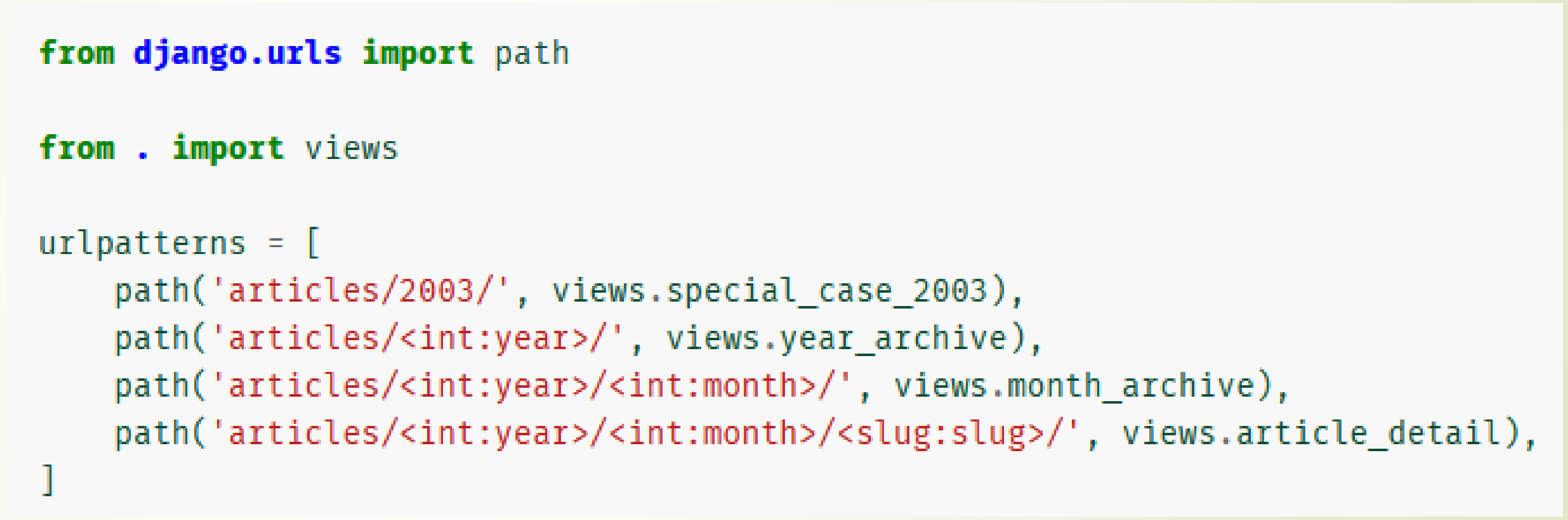
```python
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

- entry defines
  - URL-pattern
  - view function that will be called if the URL pattern is detected

Django – URL Layer

URL-Dispatcher

- example



```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

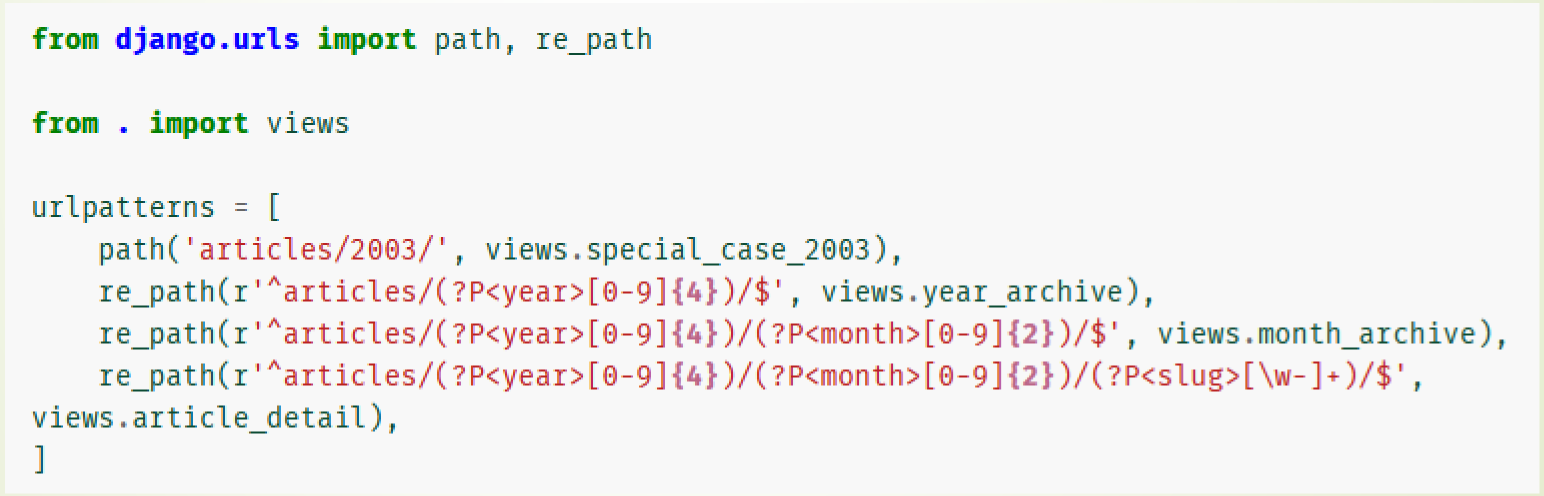
- angle brackets
  capture a value from the URL, eventually with a converter type, e.g.
  - slug   string consisting of ASCII letters or numbers, plus hyphen and underscore
  - path   non-empty string including path-separator

III.2

Django – URL Layer

URL-Dispatcher

- project-specific URLs



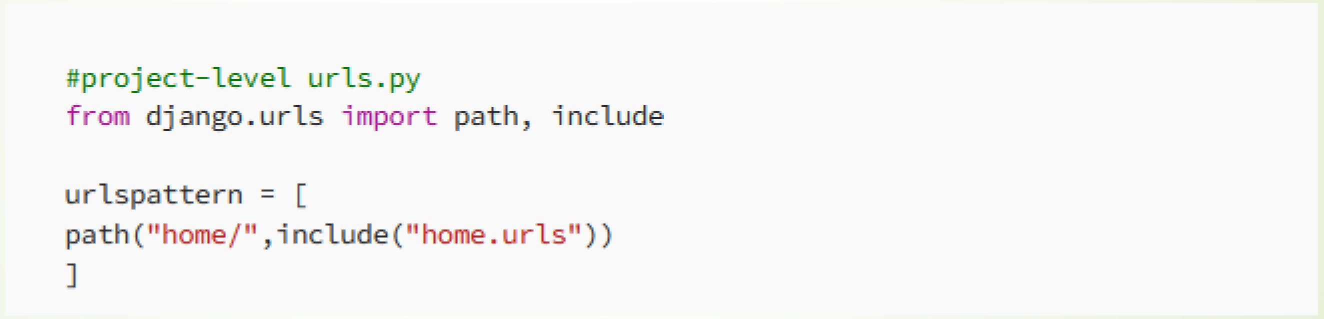
```python
from django.urls import path, re_path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    re_path(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$',
views.article_detail),
]
```

III.2

Django – URL Layer

URL-Dispatcher

- example using regular expressions
  match the incoming requests for a particular app into the app-level urls.py



```
#project-level urls.py
from django.urls import path, include

urlspattern = [
path("home/",include("home.urls"))
]
```

Django – View/Template Layer

View – function-based

- a view is a function that
  - processes an HTTP request
  - fetches the required data from the database
  - renders the data in an HTML page using an HTML template
  - returns the generated HTML in an HTTP response

```
from django.shortcuts import render

# Create your views here.
def index(request):
    # compute data to be presented
    # use objects provided by the model
    pass

    # prepare data fill the template
    htmldata = {
        "message1": "Hello",
        "message2": "CSAI-group",
    }
    # Render the HTML template index.html
     # with the data in the context variable
    return render(request, 'index.html', context=htmldata)
```

Django – View/Template Layer

Template

- philosophy
  - separate logic from presentation
  - discourage redundancy
  - be decoupled from HTML
  - assume designer competence
  - treat whitespace obviously
  - ensure safety and security
  - extensible

Django – View/Template Layer

Template

- a text file that
  - defines the structure or layout of a file (such as an HTML page)
  - written in DTL
  - uses placeholders to represent actual content
- categories

display logic	{% if %}...{% endif %}
loop control	{% for x in y %}...{% endfor %}
block declaration	{% block content %}...{% endblock %}
content import	{% include "header.html" %}
inheritance	{% extends "base.html" %}

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <h1>This is a heading</h1>
  <p>Message 1: {{ message1 }}</p>
</body>
</html>
```

Django – View/Template Layer

Template

- variables
  - are surrounded by {{ and }} and output a value from the context
- possibilities

simple variables	{{ title }}
object attributes	{{ page.title }}
dictionary lookups	{{ dict.key }}
list indexes	{{ list.0 }}
method calls	{{ var.upper }}, {{ mydict.pop }}

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <h1>This is a heading</h1>
  <p>Message 1: {{ message1 }}</p>

  <p>Message 2:{{ my_list.0 }}</p>
</body>
</html>
```

Django – View/Template Layer

Template

- tags
  - are surrounded by {% and %} and provide arbitrary logic in the generating process
  - some require beginning and ending tags

    ```
    {% tag %}
        ... tag contents ...
    {% endtag %}
    ```
  - built-in template tags

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <h1>This is a heading</h1>
  <p>Message 1: {{ message1 }}</p>

  {% if user.is_authenticated %}
     Hello, {{ user.username }}.
  {% endif %}
</body>
</html>
```

III.2

Django – View/Template Layer

Template

- filters
  - are applied using | and can modify variables for display
- possibilities

change case	{{ name | title }} or {{ units | lower }}
truncation	{{ post_content | truncatewords:50 }}
date formatting	{{ order_date | date:"D M Y" }}
list slicing	{{ list_items | slice:":3" }}
default vlues	{{ item_total | default:"nil" }}

  - built-in filters

Django – View/Template Layer

Template

- inheritance
  - create a parent template containing content shared by every page on the website and child templates that inherit these shared features from the parent
  - child templates can then add content and formatting unique to the child
  - child can extend content of parent

Django – View/Template Layer

Template

- example: base.html

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>
      {% block title %}
      {{ page_title|default:"Untitled Page" }}
      {% endblock title %}
    </title>
  </head>

  <body>
    {% block content %}
    <p>Placeholder text in base template. Replace with page content.</p>
    {% endblock content %}
  </body>
</html>
```

III.2

Django – View/Template Layer

Template

- example: mydata.html

```
{% extends 'base.html' %}

{% block title %}{{ title }}{% endblock title %}

{% block content %}
  <h1>{{ title }}</h1>
  <p>{% autoescape off %}{{ cal }}{% endautoescape %}</p>
{% endblock content %}
```

Django – Middleware Layer

Middleware

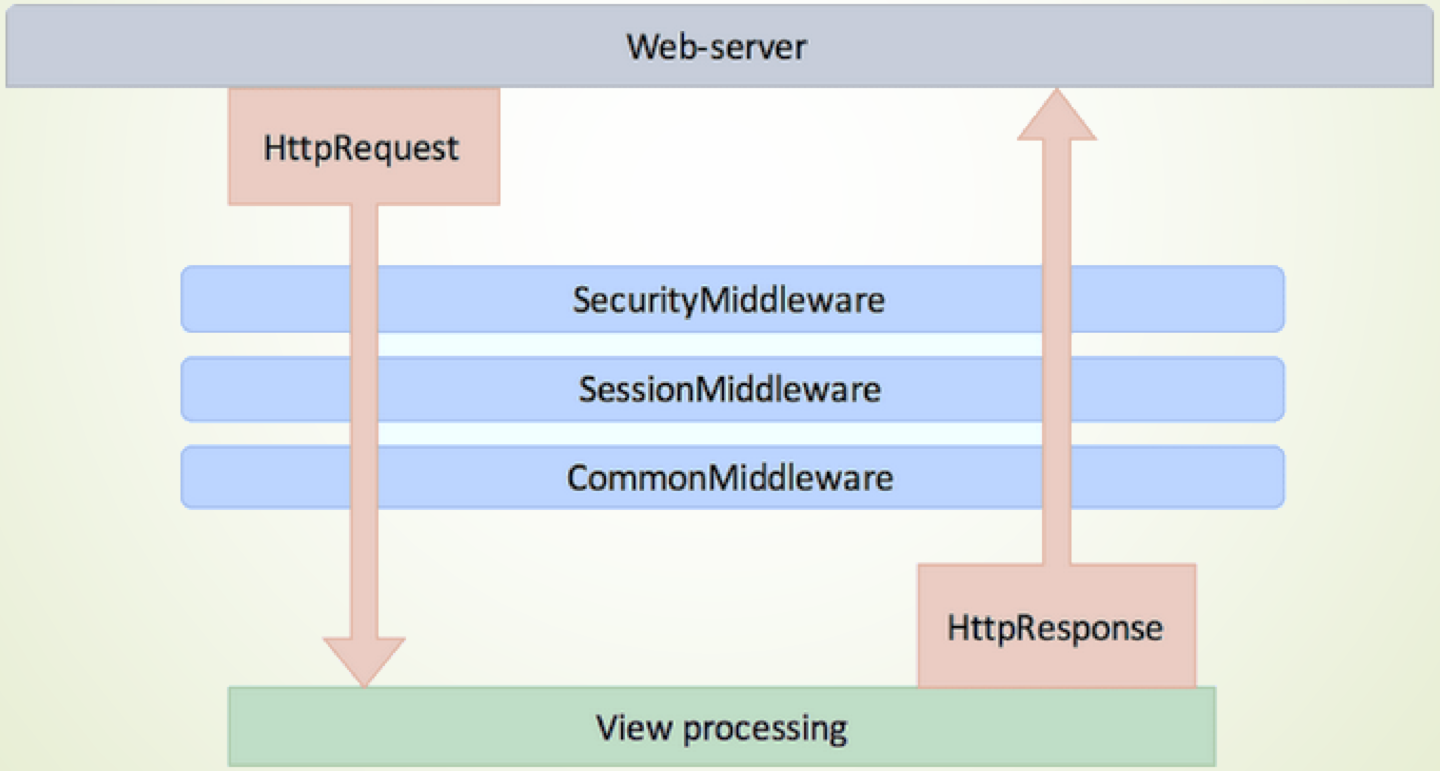
- to process requests from a browser before they reach a Django view, as well as responses from views before they reach a browser

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware']
```

- Built-in / Custom middleware

Django – Middleware Layer

Middleware



III.2

Django – Middleware Layer

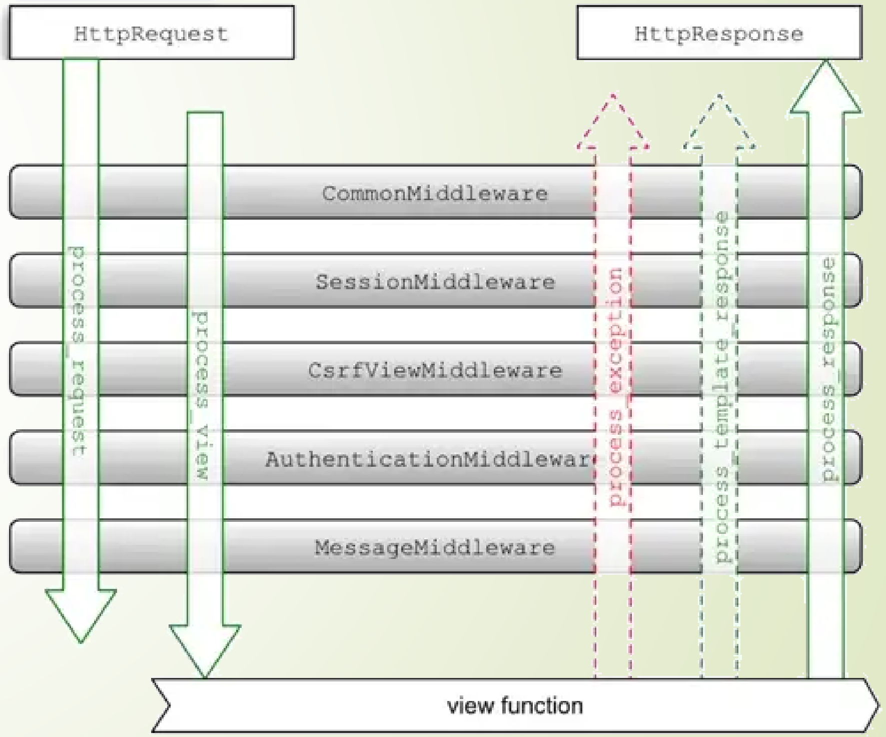
Middleware - Application

- injecting data into requests
- filtering requests
- performing
  - analytics
  - logging
  - …

Django – Middleware Layer

Middleware – Processing Steps

- builds list of methods which handle processing of request, view, response
- loops through the request methods in order
- resolves the requested URL
- loops through each of the view processing methods
- calls the view function
- loops through each of the response methods in the reverse order from request middleware



III.2

Django – Middleware Layer

Middleware – Structure

```
class ExampleMiddleware:

    def _init_(self, get_response):
        self.get_response = get_response

    def _call_(self, request):
        # Code that is executed in each request before the view is called

        response = self.get_response(request)

        # Code that is executed in each request after the view is called
        return response

    def process_view(request, view_func, view_args, view_kwargs):
        # This code is executed just before the view is called

    def process_exception(request, exception):
        # This code is executed if an exception is raised

    def process_template_response(request, response):
        # This code is executed if the response contains a render() method
        return response
```

Django – Middleware Layer

Middleware – Structure

- if middleware needs to process during request:
  - `process_request(request)`
  - `process_view(request, view_func, view_args, view_kwargs)`
- if middleware needs to process during response:
  - `process_response(request, response)`
  - `process_exception(request, exception)` - only if the view raised an exception
  - `process_template_response(request, response)` - only for template responses

Django – Session Framework

Session Framework

- session
  mechanism used for keeping track of the "state" between the site and a particular browser
- individual data items associated with the session are referenced by a "key", which is used both to store and retrieve the data
- Django uses a cookie containing a special session id to identify each browser and its associated session with the site
- the actual session data is stored in the site database by default

Django – Session Framework

Session Framework

- enabling session
- accessing the session attribute
  - within a view from the request parameter
  - is a dictionary-like object

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.check_password(request.POST['password']):
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

Django – Forms

Form handling process

- providing a framework that lets to define forms and their fields programmatically, and then use these objects to both generate the form HTML code and handle much of the validation and user interaction
- server's tasks
  - render initial form state
  - validate information sent by the browser
  - perform appropriate action

```
<form action="/csai_url/" method="post">
  <label for="team_name">Enter name: </label>
  <input
    id="team_name"
    type="text"
    name="name_field"
    value="Default name for team." />
  <input type="submit" value="OK" />
</form>
```

Django – Forms

Form handling process

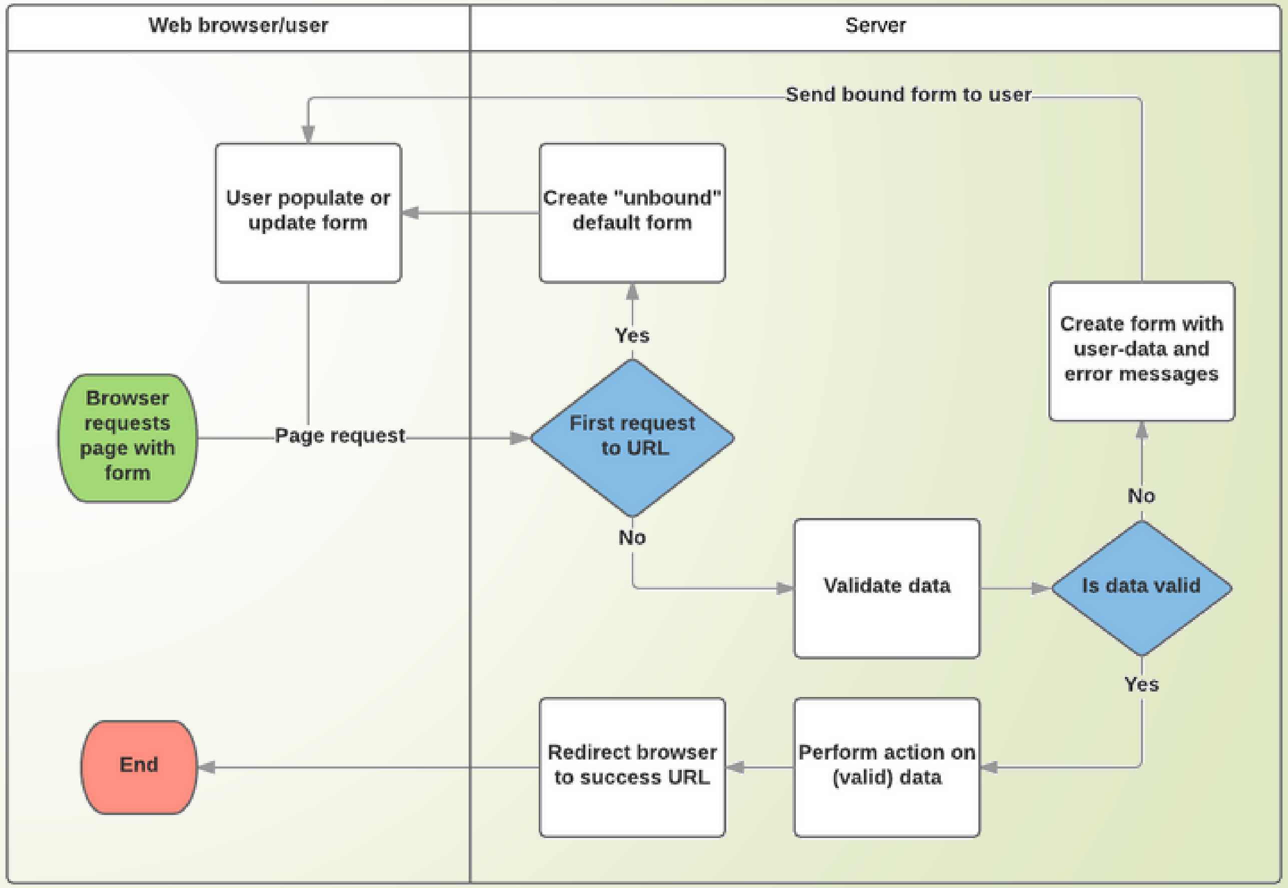
- render initial form
  - view gets a request, performs any actions required including reading data from the models
  - generates and returns an HTML page (from a template, into which we pass a context containing the data to be displayed)
- difficulty while validating information sent by the browser
  - be able to process data provided by the user, and redisplay the page if there are any errors

III.2

Django – Forms

Form handling process

1. Display the default form the first time it is requested by the user
2. Receive data from a submit request and bind it to the form
3. validate the data
4. if any data is invalid, re-display the form, this time with any user populated values and error messages
5. otherwise, perform action


Web browser/user
Server
Send bound form to user
User populate or update form
Create "unbound" default form
Yes
Create form with user-data and error messages
Browser requests page with form
Page request
First request to URL
No
No
Validate data
Is data valid
Yes
End
Redirect browser to success URL
Perform action on (valid) data

Django – Forms

Form class

```
class ContactForm(forms.Form):
      name = forms.CharField(required=False)
      email = forms.EmailField(label='Your email')

def contact(request):
    if request.method == 'POST':
        # POST, generate form with data from the request
        form = ContactForm(request.POST)
        # Reference is now a bound instance with user data sent in POST
        # process data, insert into DB, generate email, redirect to a new URL,etc
    else:
        # GET, generate blank form
        form = ContactForm()
        # Reference is now an unbound (empty) form
    # Reference form instance (bound/unbound) is sent to template for rendering
    return render(request,'about/contact.html',{'form':form})
```

Django – Forms

Form class - Validation

```
def clean_name(self):
         # Get the field value from cleaned_data dict
         value = self.cleaned_data['name']
         # Check if the value is all upper case
         if value.isupper():
            # Value is all upper case, raise an error
            raise forms.ValidationError("""Please don't use all upper
                 case for your name, use lower case""",code='uppercase')
         # Always return value
         return value
```

Django – Forms

Form class - Template

```
<form method="POST">
  {% csrf_token %}

  <table>
    {{form.as_table}}
  </table>
  <input type="submit" value="Submit form">

</form>
```