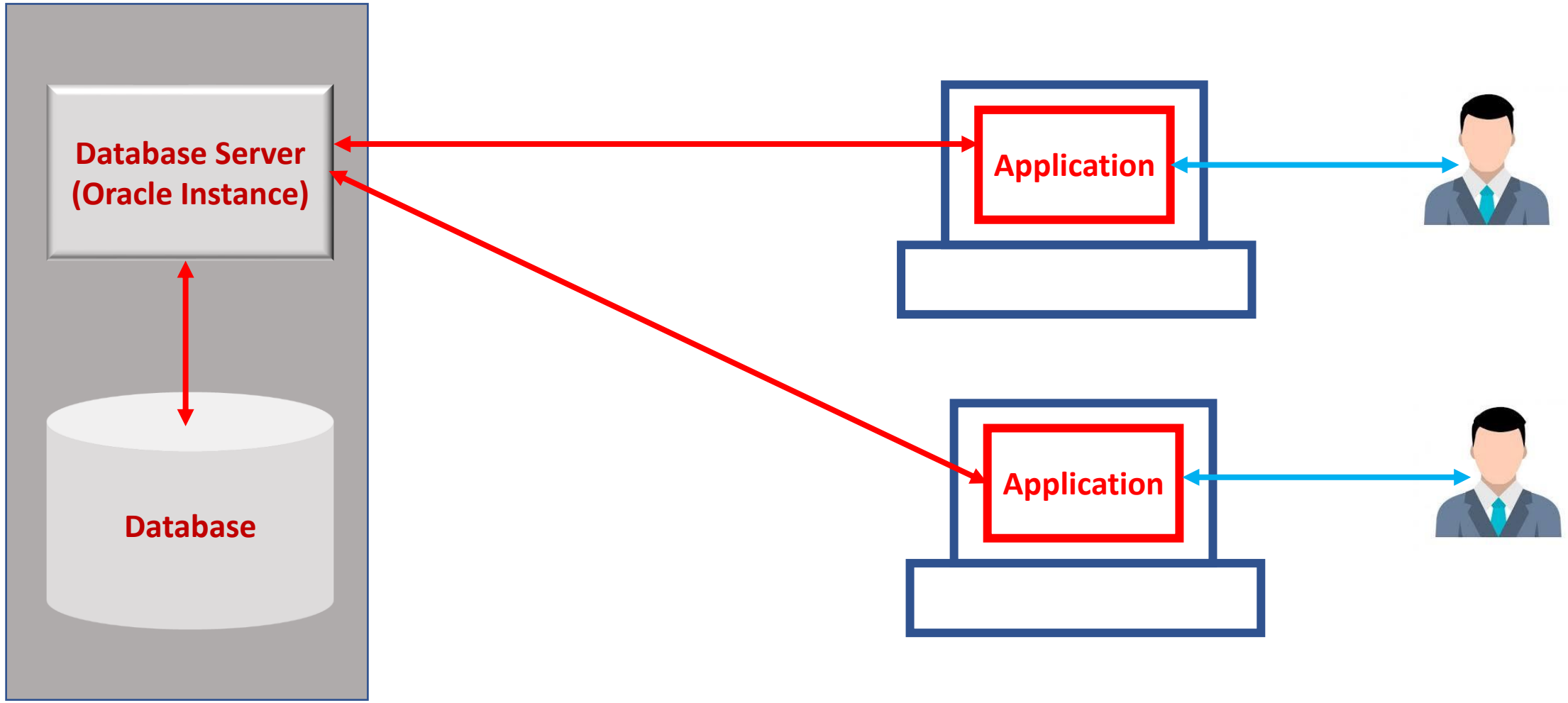
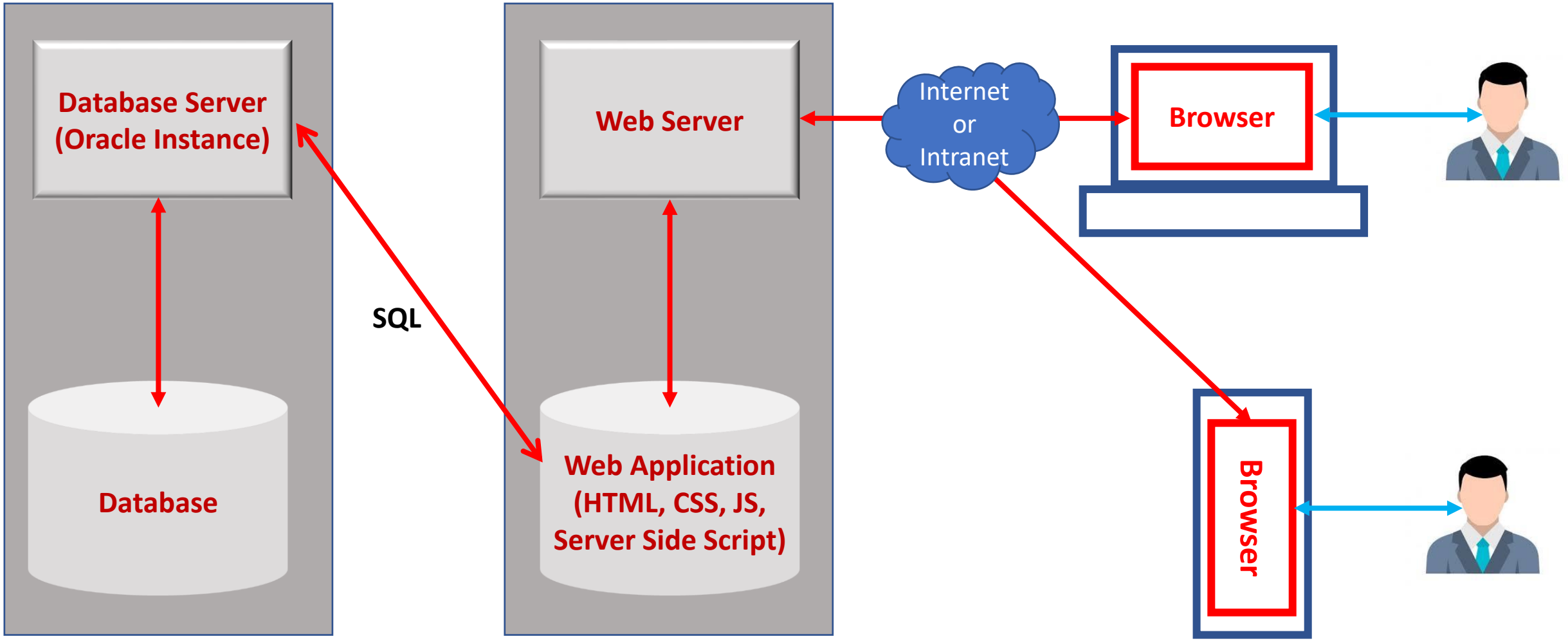
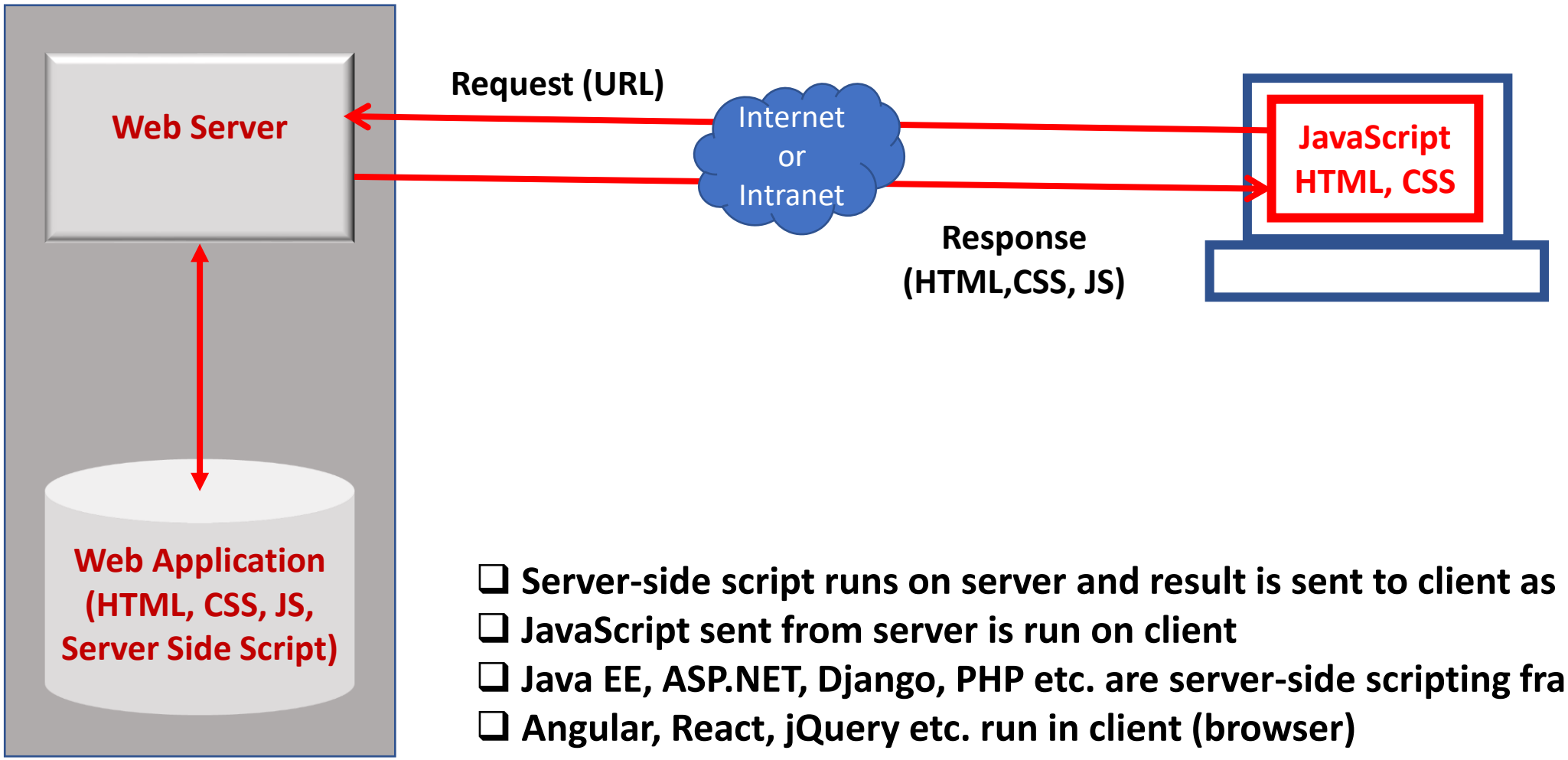
Client Server Architecture



Web Application Architecture



Server-side Scripting (back-end) vs. Client-side Scripting (front-end)



- Server-side script runs on server and result is sent to client as HTML
- JavaScript sent from server is run on client
- Java EE, ASP.NET, Django, PHP etc. are server-side scripting frameworks
- Angular, React, jQuery etc. run in client (browser)

What is Django?

- It is a framework to build web applications.
- Django takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel.
- Tag line is - The web framework for ***perfectionists with deadlines***.
- Django **3.1.1** is the latest version, which was released in **Sep, 2020**.
- Official website is: **https://www.djangoproject.com**

Why Django?

- It provides a set of tools and functionalities that solve many common problems associated with Web development, such as security features, database access, sessions, template processing, URL routing, and much more.
- It’s definitely the most complete, offering a wide range of features out-of-the-box, such as a standalone Web server for development and testing, caching, middleware system, ORM, template engine, form processing.
- It’s under active development for more than 12 years now, proving to be a mature, reliable and secure Web framework.
- Websites like Instagram, Mozilla use Django.
- Get stats at **djangosites.org**

Installing Django

- Django is provided as a package and installed using PIP.
- Django also comes with its own lightweight web server for testing our applications.
- It is recommended to install Django in virtual environment, though it is not mandatory.

```
>pip install django
Collecting django
  Downloading Django-3.1.1-py3-none-any.whl (7.8 MB)
     |████████████████████████████████| 7.8 MB 1.6 MB/s
Requirement already satisfied: asgiref~=3.2.10 in c:\python\lib\site-packages (from
django) (3.2.10)
Requirement already satisfied: pytz in c:\python\lib\site-packages (from django) (2019.3)
Requirement already satisfied: sqlparse>=0.2.2 in c:\python\lib\site-packages (from
django) (0.3.1)
Installing collected packages: django
Successfully installed django-3.1.1
```

Verifying Django Installation

After the installation has completed, you can verify your Django installation by running django-admin as follows:

```
>django-admin --version
3.1.1
```

NOTE: django-admin.exe is installed into python/scripts folder. If /python/scripts is not in system PATH then go into /python/scripts folder and run django-admin.exe

Creating a project

- A django project is a collection of applications.
- First we need to create a project and then create applications inside project.
- Use django-admin to create django project as shown below after getting into folder where we want to create project folder:

```
>django-admin startproject website
```

When you create a project, a folder with the same name as project is created. Inside that another folder is created with following files:

```
website/
    manage.py
    website/
        asgi.py
        settings.py
        urls.py
        wsgi.py
        __init__.py
```

Files in Project

manage.py	A command-line utility that lets you interact with this Django project in various ways.
website	This directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. website.urls).
__init__.py	An empty file that tells Python that this directory should be considered a Python package.
settings.py	Settings/configuration for this Django project.
urls.py	The URL declarations for this Django project; a "table of contents" of your Django powered site.
wsgi.py	An entry-point for WSGI (Web Server Gateway Interface)-compatible web servers to serve your project.
asgi.py	An entry-point for ASGI(Application Server Gateway Interface)-compatible web servers to serve your project.

Running Development Server (Web Server)

- Django provides Development Server that runs by default at port number 8000 on current system 127.0.0.1.
- Get into website (outer) folder where manage.py is present and start server as follows:

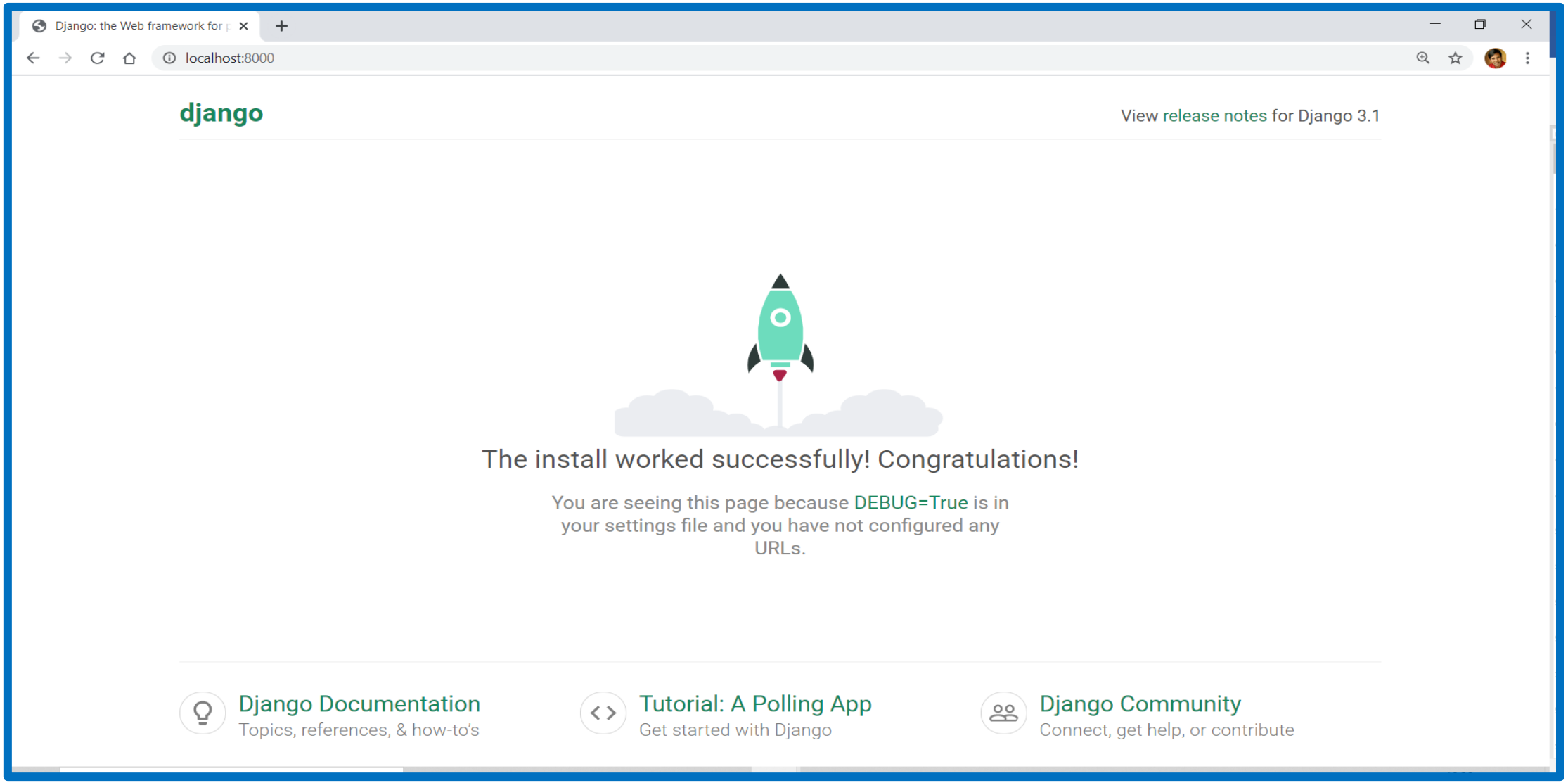
```
>python manage.py runserver
Watching for file changes with StatReloader
.
.
.
Django version 3.1.1, using settings 'website.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

We can provide port number (8888) of the server at the time of running server as follows:

```
>python manage.py runserver 8888
```

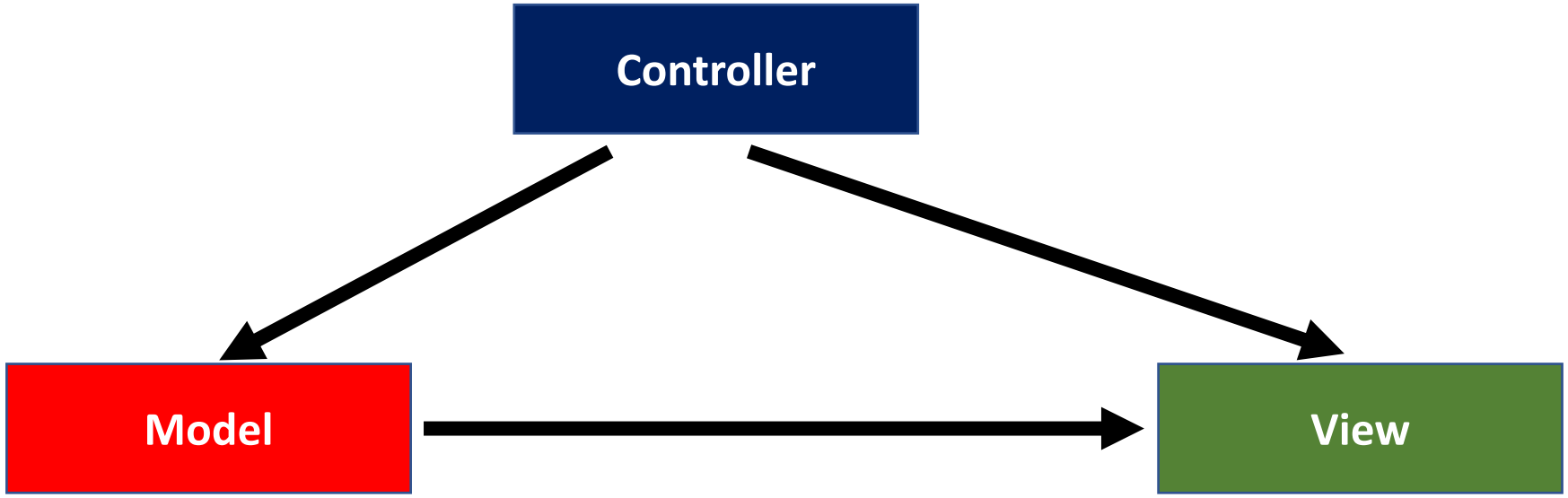
Test Your Project

- Go to browser and invoke **http://127.0.0.1:8000** or **http://localhost:8000**



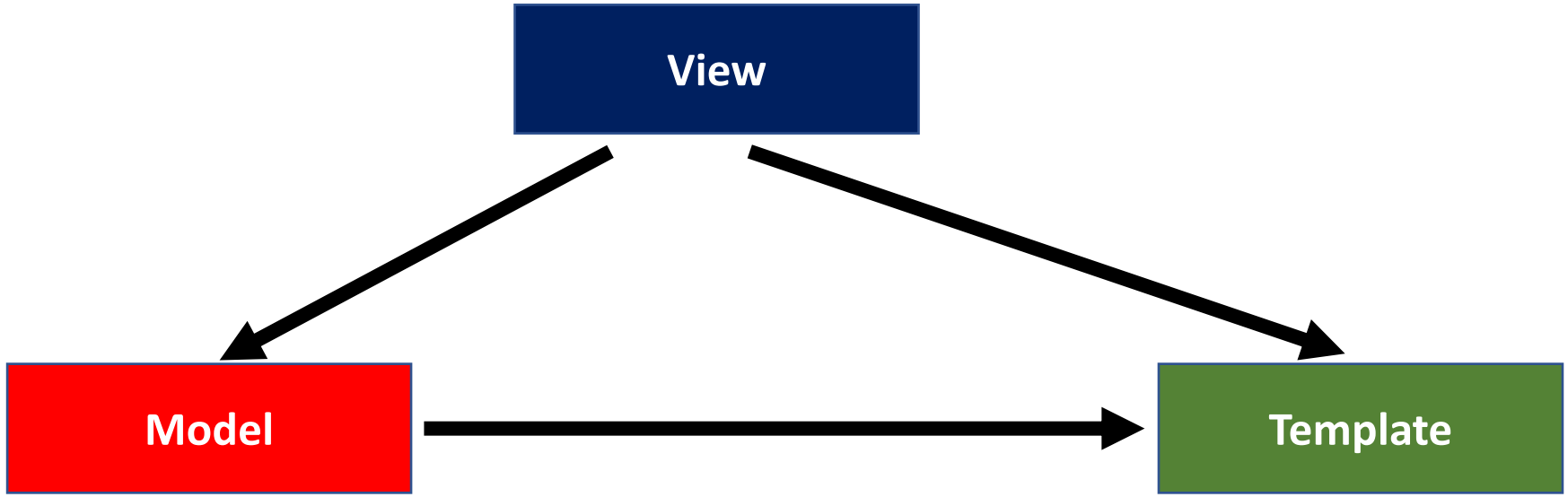
MVC (Model View Controller)

- Model is a class that represents data.
- View is to present data that is in model.
- Controller is invoked by URL. It uses model and invokes view.



MVT (Model View Template)

- Model is a class that represents data.
- View is a function or a class that is invoked by URL and it handles business logic. Invokes template and passes data to template.
- Template presents data using Django Template Language.



Project vs. Application

- An application is a web application that does something.
- A project is a collection of settings and applications for a particular website.
- A project can contain multiple applications.
- An application can be in multiple projects.
- We can’t run an application without a project.

Creating an Application

- An application is the actual container of views, models and templates.
- An application is part of a project.
- Get into project folder and create an application as shown below:

```
>python manage.py startapp basics
```

It creates folder **basics** with the following files in that folder:

```
basics/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Files in Application

migrations (folder)	Django stores some files (migration scripts) to keep track of the changes you make to models in **models.py** file, so as to keep the database and the models.py synchronized. This folder contains migration scripts.
admin.py	This is a configuration file for a built-in Django app called Django Admin.
apps.py	This is a configuration file of the app itself.
models.py	This is where we define the entities of our Web application. The models are translated automatically by Django into database tables.
tests.py	This file is used to write unit tests for the app.
views.py	This is the file where we handle the request/response cycle of our Web application.

Add Application to Project

- Every application must be associated with a project.
- Add application name to INSTALLED_APPS list in **settings.py** file in project folder.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'basics',
]
```

Creating First View

- A view is a function that is invoked when associated URL is invoked from client.
- It takes request objects (HttpRequest), which maps to HTTP request and contains information about client, as parameter.
- It must return response object (HttpResponse), which contains response to be sent back to client.

basics\views.py

```
from django.shortcuts import render
from django.http import HttpResponse


def welcome(request):
    return HttpResponse("<h1>Welcome To Django</h1>")
```

Linking view to URL

- A view is a function that is executed when a URL is requested from browser
- We need to associate views with URLs using urls.py module

website\urls.py

```
from django.contrib import admin
from django.urls import path
import basics.views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('welcome/', basics.views.welcome),
]
```

- Go to browser and invoke **http://127.0.0.1:8000/welcome**

Separating URLs related to an Application

- It is recommended to provide URLs related to an application using **urls.py** module in application.
- Redirect all requests related to application to **urls.py** module of application using **include**() function.

website\urls.py

```
from django.contrib import admin
from django.urls import path, include
import basics.views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('welcome/', basics.views.welcome),
    path('basics/', include('basics.urls')),
]
```

Create urls.py in Application

- Copy **urls.py** from website folder to basics folder and modify it

basics\urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('welcome/', views.welcome),
]
```

- Go to browser and invoke **http://127.0.0.1:8000/basics/welcome**

path()

The path() function is passed four arguments, two required: route and view, and two optional: kwargs, and name.

Parameter	Meaning
route	Route is a string that contains a URL pattern. When processing a request, Django starts at the first pattern in urlpatterns and makes its way down the list, comparing the requested URL against each pattern until it finds one that matches. Patterns don't search GET and POST parameters, or the domain name.
view	When Django finds a matching pattern, it calls the specified view function with an *HttpRequest* object as the first argument and any "captured" values from the route as keyword arguments.
kwargs	Arbitrary keyword arguments can be passed in a dictionary to the target view.
name	Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.

HttpRequest object

- HttpRequest object represents a request coming from client.
- It provides information about requests such as Http method, cookies and request parameters.

Member	Meaning
scheme	A string representing the scheme of the request (http or https usually).
body	The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms binary images, XML payload etc.
path	A string representing the full path to the requested page, not including the domain.
method	A string representing the HTTP method used in the request.
GET	A dictionary-like object containing all given HTTP GET parameters.
POST	A dictionary-like object containing all given HTTP POST parameters, providing that the request has form data.
COOKIES	A standard Python dictionary containing all cookies. Keys and values are strings.

HttpRequest object

Member	Meaning
FILES	A dictionary-like object containing all uploaded files. Each key in FILES is the name from the <input type="file" name="" />. Each value in FILES is an UploadedFile.
META	A standard Python dictionary containing all available HTTP headers.
user	An object of type AUTH_USER_MODEL representing the currently logged-in user. If the user isn't currently logged in, user will be set to an instance of django.contrib.auth.models.AnonymousUser.
session	A readable-and-writable, dictionary-like object that represents the current session. This is only available if your Django installation has session support activated.
get_full_path()	Returns the path, plus an appended query string, if applicable.
is_secure()	Returns True if the request is secure; that is, if it was made with HTTPS.
is_ajax()	Returns True if the request was made via an XMLHttpRequest, by checking the HTTP_X_REQUESTED_WITH header for the string "XMLHttpRequest".

HttpResponse object

- In contrast to HttpRequest objects, which are created automatically by Django, HttpResponse objects are to be created by us.
- Each view you write is responsible for instantiating, populating and returning an HttpResponse.

Attribute	Meaning
content	A byte string representing the content, encoded from a Unicode object if necessary.
status_code	The HTTP status code for the response.
reason_phrase	The HTTP reason phrase for the response.

HttpResponse object

Method	Meaning
setdefault()	Sets a value for a header.
set_cookie()	Sends a cookie to client.
delete_cookie()	Deletes the cookie with the given key. Fails silently if the key doesn’t exist.
write(content)	Writes given content to client.
writelines(lines)	Writes a list of lines to the response. Line separators are not added. This method makes an HttpResponse instance as a stream-like object.

Passing data from Client

- It is possible to send data to server at the time of making a request through request parameters.

http://127.0.0.1:8000/basics/wish**?name=Srikanth**

basics\views.py

```
def wish(request):
    name = request.GET['name']
    return HttpResponse(f"<h1>Hello, {name}</h1>")
```

Templates

- A template is HTML page that can use Django Template language constructs.
- A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template.
- The Django template system provides tags which function similar to some programming constructs – an **if** tag for boolean tests, a **for** tag for looping, etc.
- Templates are **.html** files stored in **templates** folder in application.

Variables in Templates

- Variables are enclosed in {{ }}.
- When the template engine encounters a variable, it evaluates that variable and replaces it with the result.
- Variable names consist of any combination of alphanumeric characters and the underscore ("_").

```
{{message}}
```

Filters in Templates

- You can modify variables for display by using filters.
- Use a pipe (|) to apply a filter.
- Filters look like this: **{{ name | lower }}.** This displays the value of the {{ name }} variable after being filtered through the lower filter, which converts text to lowercase.
- Filters can be “chained.” The output of one filter is applied to the next.
- Expression **{{ text|escape|linebreaks }}** is a common idiom for escaping text contents, then converting line breaks to <p> tags.
- Some filters take arguments. A filter argument looks like this: **{{ amount | floatformat: 2}}** will display amount with 2 decimal places.
- Filter arguments that contain spaces must be quoted.

```
{{expression  | filter}}
```

Built-in Filters

Filter	What it does?
capfirst	Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.
center	Centers the value in a field of a given width.
default	If value evaluates to False, uses the given default. Otherwise, uses the value. Ex. {{value|default:"nothing"}}
escape	Escapes a string's HTML. Specifically, it makes these replacements.
floatformat	When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed.
length	Returns the length of the value. This works for both strings and lists.
linebreaks	Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break ( ) and a new line followed by a blank line becomes a paragraph break (</p>).
linenumbers	Displays text with line numbers.
ljust	Left-aligns the value in a field of a given width.
lower	Converts a string into all lowercase.
upper	Converts a string into all uppercase.
urlencode	Escapes a value for use in a URL.
wordcount	Returns the number of words.
wordwrap	Wraps words at specified line length.

Django Tags in Templates

- Tags look like this: {% tag %}.
- Tags are more complex than variables.
- Some tags create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.
- Django ships with about two dozen built-in template tags.

```
{% tag %}
```

```
<ul>
  {% for name in names %}
      <li>{{name}}</li>
  {% endfor %}
</ul>
```

Built-in Tags

Tag	Meaning
block	Defines a block that can be overridden by child templates.
comment	Ignores everything between {% comment %} and {% endcomment %}.
csrf_token	This tag is used for CSRF protection.
extends	Signals that this template extends a parent template.
for	Loops over each item in an array, making the item available in a context variable.
if	The {% if %} tag evaluates a variable, and if that variable is true (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output.
ifequal	Output the contents of the block if the two arguments equal each other.
ifnotequal	Just like ifequal, except it tests that the two arguments are not equal.
include	Loads a template and renders it with the current context. Ex: {% include "foo/bar.html" %}

HTML Form

- HTML form is used to take input from user using INPUT elements.
- Tag <form> is used to create a form.
- Attribute action is used to specify the URL to be invoked when form is submitted.
- Data sent from form is to be accessed using request.GET or request.POST attributes in view.
- GET method (default) sends data along with URL (Query String).
- POST method (method ='post') sends data along with body of the request.

HTML Form with GET Method

basics\templates\interest.html

```
<h2>Interest Calculation </h2>
<form>
  Deposit Amount<br/>
  <input type='number' name='amount' />
  <p></p>

  Interest Rate<br/>
  <input type='number' name='rate' step='any' />
  <p></p>
  <input type='submit' value="Calculate"/>
</form>

{% if interest %}
  <h3>Interest  = {{interest}} </h3>
{% endif %}
```

HTML Form with GET Method

basics\views.py

```
def interest(request):
  if 'amount' in request.GET:
     amount = float(request.GET['amount'])
     rate = float(request.GET['rate'])
     interest = amount * rate / 100
     return render(request,'interest.html', {'interest' : interest })
  else:
     return render(request,'interest.html')
```

basics\urls.py

```
…
urlpatterns = [
    path('interest/', views.interest),
]
```

HTML Form with POST Method

basics\templates\interest_post.html

```
<h2>Interest Calculation </h2>
<form method="post">
  {% csrf_token %}
  Deposit Amount<br/>
  <input type='number' name='amount' />
  <p></p>

  Interest Rate<br/>
  <input type='number' name='rate' />
  <p></p>
  <input type='submit' value="Calculate"/>
</form>

{% if interest %}
  <h3>Interest  = {{interest}}  </h3>
{% endif %}
```

HTML Form with POST Method

basics\views.py

```
def interest_post(request):
  if request.method == "POST":
     amount = float(request.POST['amount'])
     rate = float(request.POST['rate'])
     interest = amount * rate / 100
     return render(request,'interest.html', {'interest' : interest })
  else:   # GET request
     return render(request,'interest.html')
```

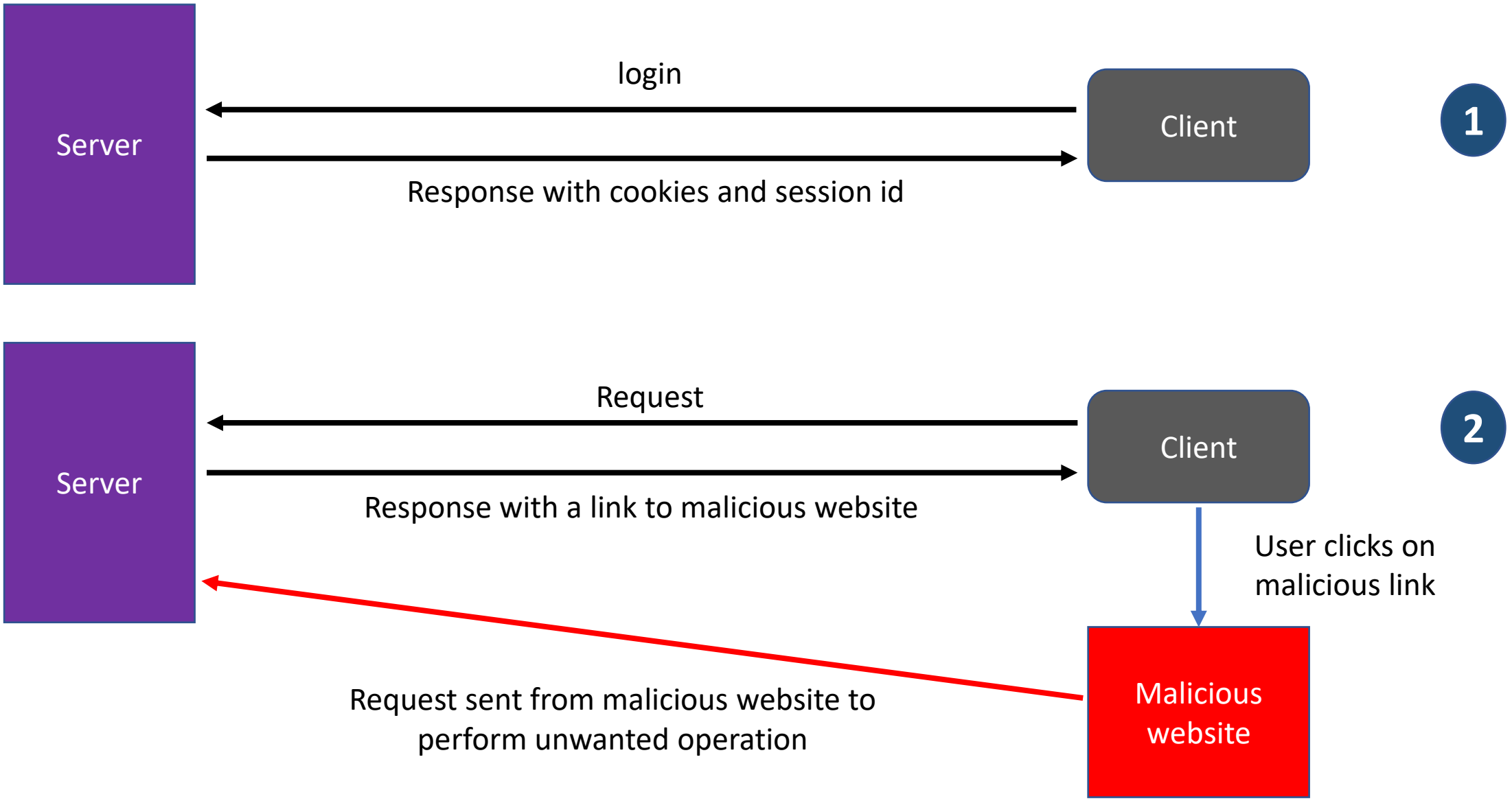
basics\urls.py

```
…
urlpatterns = [
    path('interest_post/', views.interest_post),
]
```

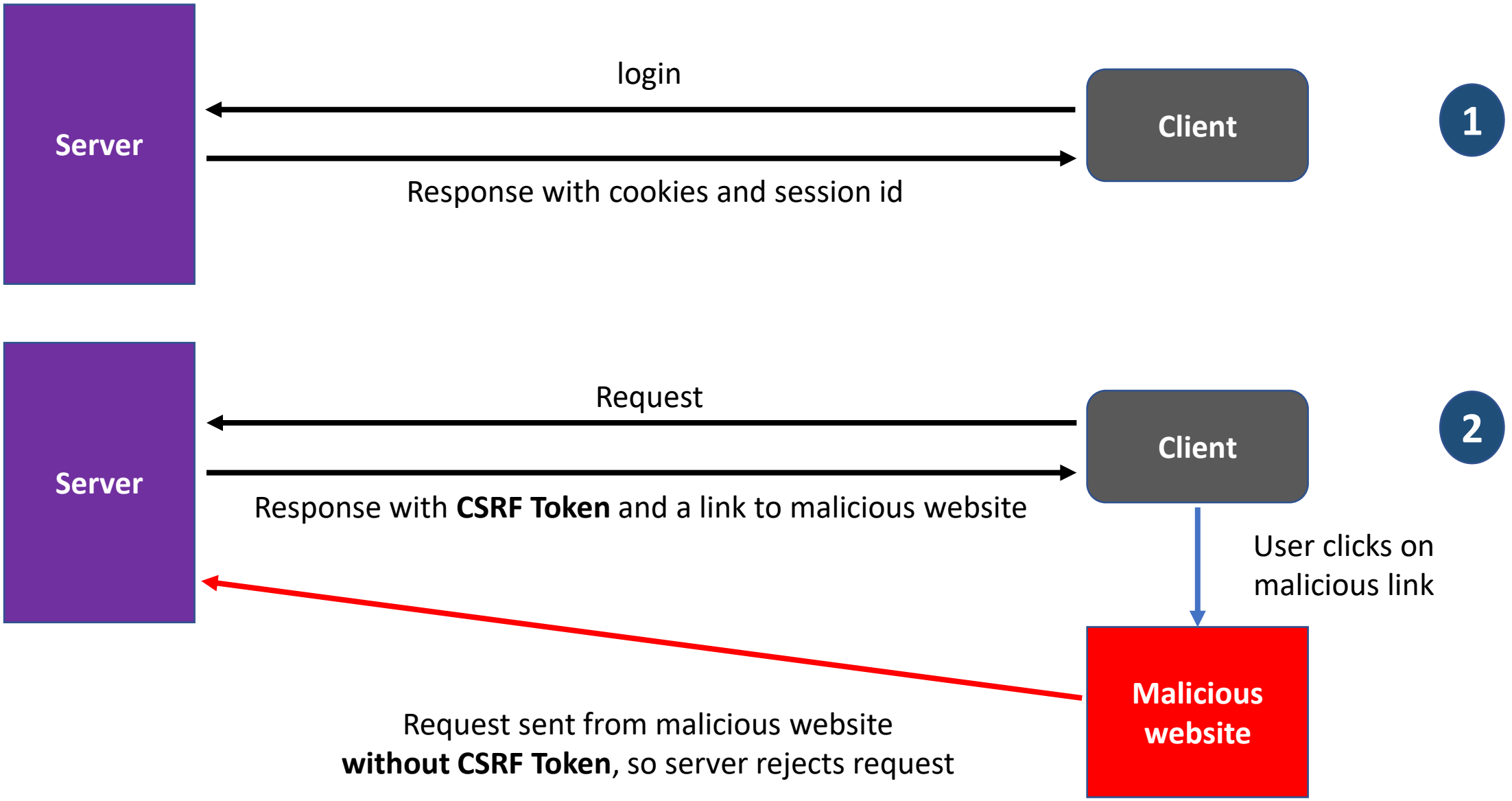
CSRF (Cross-site Request Forgery)

- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
- The CSRF middleware is activated by default in the MIDDLEWARE setting of the project.
- In any template that uses a POST form, use the **csrf_token** tag inside the <form> element if the form is for an internal URL.
- When csrf_token tag is used it sends a token that is to be sent back to server client to make a valid POST request.
- Token is a hidden field in HTML form.

Without CSRF Token



With CSRF Token



Database Access using DB-API 2.0

- Connect to Database using sqlite3 module
- Use Connection object to create Cursor
- Use Cursor to manipulate Database

```
CREATE TABLE authors (
    auid INTEGER PRIMARY KEY AUTOINCREMENT,
    fullname VARCHAR (30),
    email VARCHAR (50) UNIQUE,
    mobile VARCHAR (10)
);
```

Retrieving authors from database

```
import sqlite3

def list_authors(request):
    try:
        con = sqlite3.connect("db.sqlite3")
        cur = con.cursor()
        cur.execute("select * from authors")
        authors = cur.fetchall()
        con.close()
        return render(request, 'db/list_authors.html', {'authors': authors})
    except Exception as ex:
        print("Error :", ex)
        return render(request, 'db/list_authors.html')
```

Display details of Authors

```
<h2>Authors</h2>
{% if authors %}
  <table cellpadding="5px" border="1">
    <tr> <th>Id</th> <th>Name</th> <th>Email</th> <th>Mobile</th> </tr>
    {% for author in authors %}
      <tr>
        <td>{{author.0}}</td>
        <td>{{author.1}}</td>
        <td>{{author.2}}</td>
        <td>{{author.3}}</td>
      </tr>
    {% endfor %}
</table>
{% else %}
    <h3>Sorry! Could not retrieve details of authors!</h3>
{% endif %}
```

Django Form

- Forms are used to take input from user in web applications.
- Django forms can make use of model in application and generate required HTML and send it to client.
- A Django form contains fields to manifest into HTML elements to interact with users. These fields are themselves classes; they manage form data and perform validation when a form is submitted.
- Forms receive submitted data from client and make it available for process.

django.forms.Form class

- A form is a class that is derived from **Form** class.
- It defines one or more fields, where each field can have its own attributes related to presentation and validation.
- By default form fields are not bound to data. We need to explicitly bind data to form fields at the time of creating form.
- We can check whether form is bound or not using **is_bound** attribute.
- Method **is_valid**() is used to check whether data in the form is valid.
- If data is valid then form's data is placed in **cleaned_data** attribute.
- A form, when rendered doesn't provide <form> tag and submit button. So, we need to make sure we provide them in template.
- Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as <input type="text">.

Members of Form class

Member	Meaning
is_bound	If you need to distinguish between bound and unbound form instances at runtime, check the value of the form's is_bound attribute.
clean()	Implement a clean() method on your Form when you must add custom validation for fields that are interdependent.
is_valid()	The primary task of a Form object is to validate data. With a bound Form instance, call the is_valid() method to run validation and return a boolean designating whether the data was valid.
errors	Access the errors attribute to get a dictionary of error messages. In this dictionary, the keys are the field names, and the values are lists of strings representing the error messages. The error messages are stored in lists because a field can have multiple error messages. You can access errors without having to call is_valid() first. The form's data will be validated the first time either you call is_valid() or access errors.
changed_data	The changed_data attribute returns a list of the names of the fields whose values in the form's bound data (usually request.POST) differ from what was provided in initial.
fields	You can access the fields of Form instance from its fields attribute.
cleaned_data	Each field in a Form class is responsible not only for validating data, but also for "cleaning" it – normalizing it to a consistent format.

Django Shell

- Invokes interactive Python shell
- Make Django API available to shell

```
>python manage.py shell
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC
v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
(InteractiveConsole)
>>>
```

Django Form class

basics\forms.py

```
from django import forms

class InterestForm(forms.Form):
     amount = forms.FloatField(label='Amount')
     rate = forms.FloatField(label='Interest Rate')
```

basics\urls.py

```
…
urlpatterns = [
    path('interest_form/', views.interest_form),
]
```

Template for Django Form

basics\templates\interest_form.html

```
<h2>Interest Calculation</h2>
<form action="interest_form" method="post">
  {% csrf_token %}
  <table>
     {{form}}
  </table>
  <p></p>
  <input type='submit' value="Calculate"/>
</form>

{% if interest %}
  <h3>Interest  = {{interest}} </h3>
{% endif %}
```

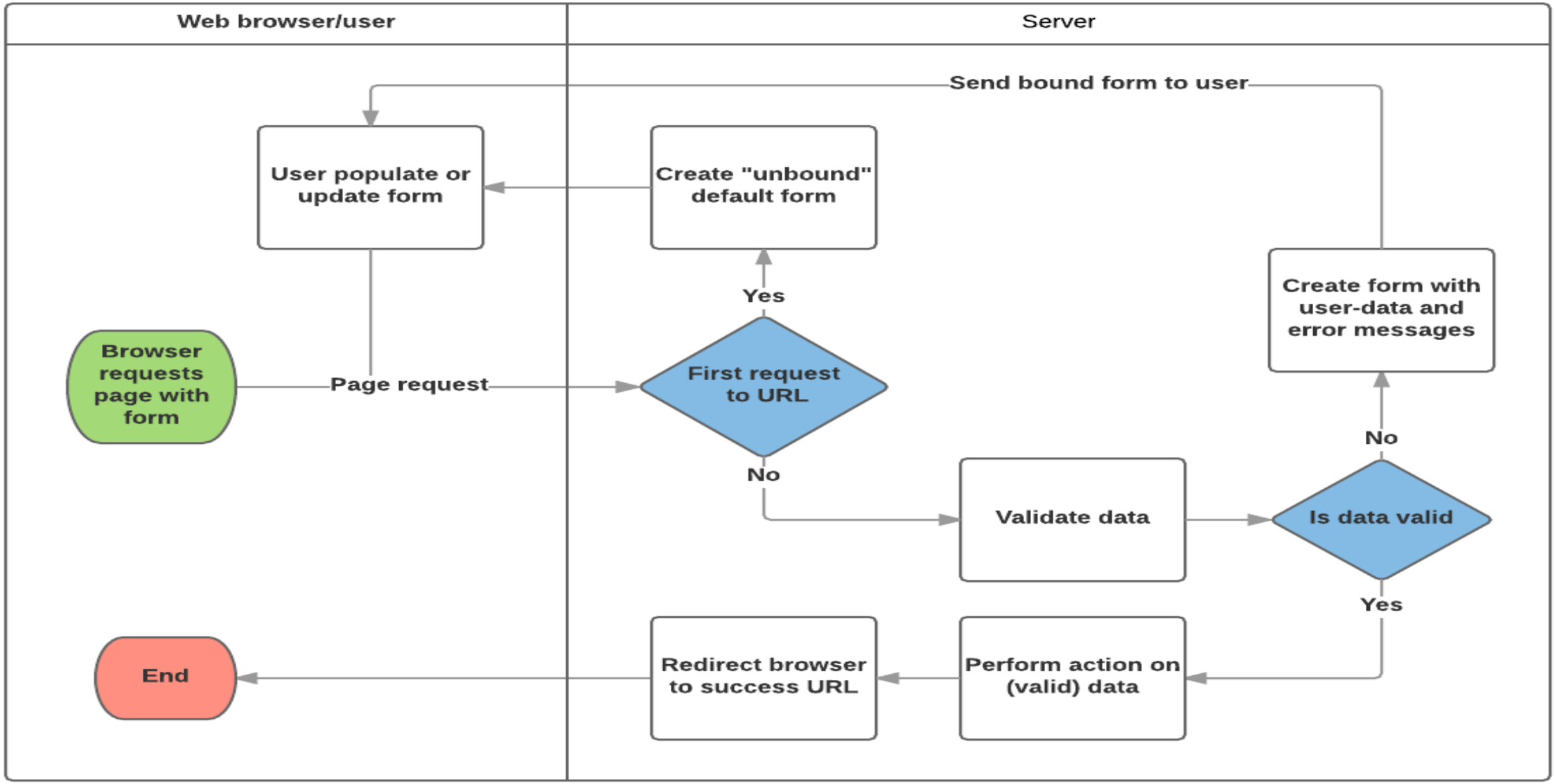
Processing Django Form

basics\views.py

```
from .forms import InterestForm

def interest_form(request):
  if request.method == "POST":
     form = InterestForm(request.POST)  # bind POST data to form
     amount = float(form.cleaned_data['amount'])
     rate = float(form.cleaned_data['rate'])
     interest = amount * rate / 100
     return render(request,'interest_form.html',
                     {'form' : form, 'interest' : interest })
  else:   # GET request
     form = InterestForm()
     return render(request,'interest_form.html',{'form' : form})
```

Form Request Life Cycle



BoundField Attributes

Each field of the form is of type **BoundField**. It has the following attributes:

Attribute	Meaning
auto_id	The HTML ID attribute for this returns an empty string if Form.auto_id is False.
data	This property returns the data for this BoundField extracted by the widget's value_from_datadict() method, or None if it wasn't given.
errors	A list-like object that is displayed as an HTML <ul class="errorlist"> when printed.
form	The Form instance this BoundField is bound to.
help_text	The help_text of the field.
html_name	The name that will be used in the widget's HTML name attribute. It takes the form prefix into account.
id_for_label	Use this property to render the ID of this field.
is_hidden	Returns True if this BoundField's widget is hidden.
label	The label of the field. This is used in label_tag().
name	The name of this field in the form.

BooleanField

- Default widget: CheckboxInput
- Empty value: False
- Normalizes to: A Python True or False value
- Validates that the value is True (e.g. the check box is checked) if the field has required=True
- Error message key: required

CharField

- Default widget: TextInput
- Empty value: Whatever you've given as empty_value
- Normalizes to: A string
- Error message keys: required, max_length, min_length
- Has three optional arguments for validation: max_length, min_length, strip
- Validates max_length or min_length, if they are provided. Otherwise, all inputs are valid.

ChoiceField

- Default widget: Select
- Empty value: '' (an empty string)
- Normalizes to: A string
- Validates that the given value exists in the list of choices
- Error message keys: required, invalid_choice
- The invalid_choice error message may contain %(value)s, which will be replaced with the selected choice
- Choices is either an iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field, or a callable that returns such an iterable. This argument accepts the same formats as the choices argument to a model field. Defaults to an empty list.

DateField

- Default widget: DateInput
- Empty value: None
- Normalizes to: A Python datetime.date object
- Validates that the given value is either a datetime.date, datetime.datetime or string formatted in a particular date format
- Error message keys: required, invalid
- Takes one optional argument: input_formats - A list of formats used to attempt to convert a string to a valid datetime.date object

RegexField

- Default widget: TextInput
- Empty value: '' (an empty string)
- Normalizes to: A string
- Validates that the given value matches against a certain regular expression
- Error message keys: required, invalid
- Takes one required argument: regex
- A regular expression specified either as a string or a compiled regular expression object
- Also takes max_length, min_length, and strip, which work just as they do for CharField

IntegerField

- Default widget: NumberInput when Field.localize is False, else TextInput
- Empty value: None
- Normalizes to: A Python integer
- Validates that the given value is an integer. Uses MaxValueValidator and MinValueValidator if max_value and min_value are provided. Leading and trailing whitespace is allowed, as in Python's int() function
- Error message keys: required, invalid, max_value, min_value
- The max_value and min_value error messages may contain %(limit_value)s, which will be substituted by the appropriate limit

FloatField

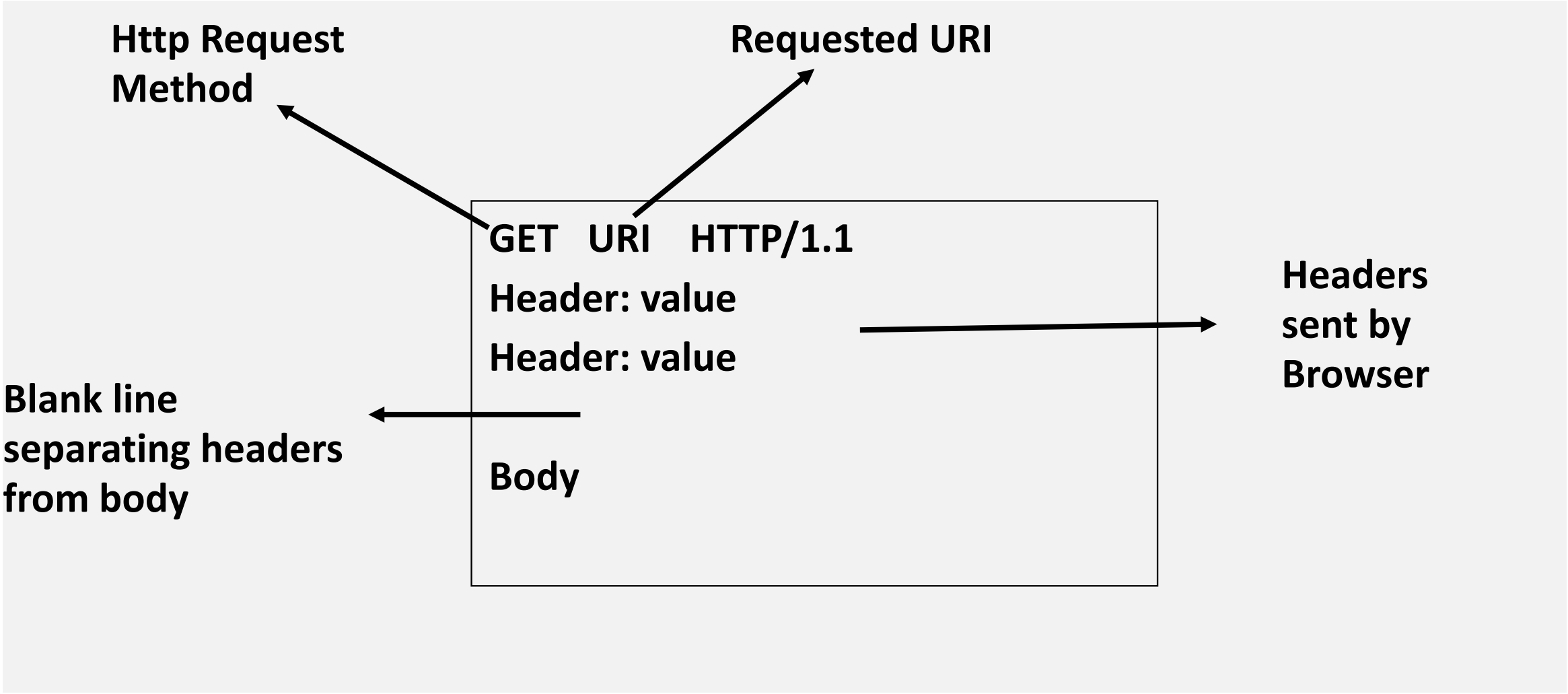
- Default widget: NumberInput when Field.localize is False, else TextInput
- Empty value: None
- Normalizes to: A Python float
- Validates that the given value is a decimal. Uses MaxValueValidator and MinValueValidator if max_value and min_value are provided. Leading and trailing whitespace is ignored
- Error message keys: required, invalid, max_value, min_value

Rendering Fields Manually

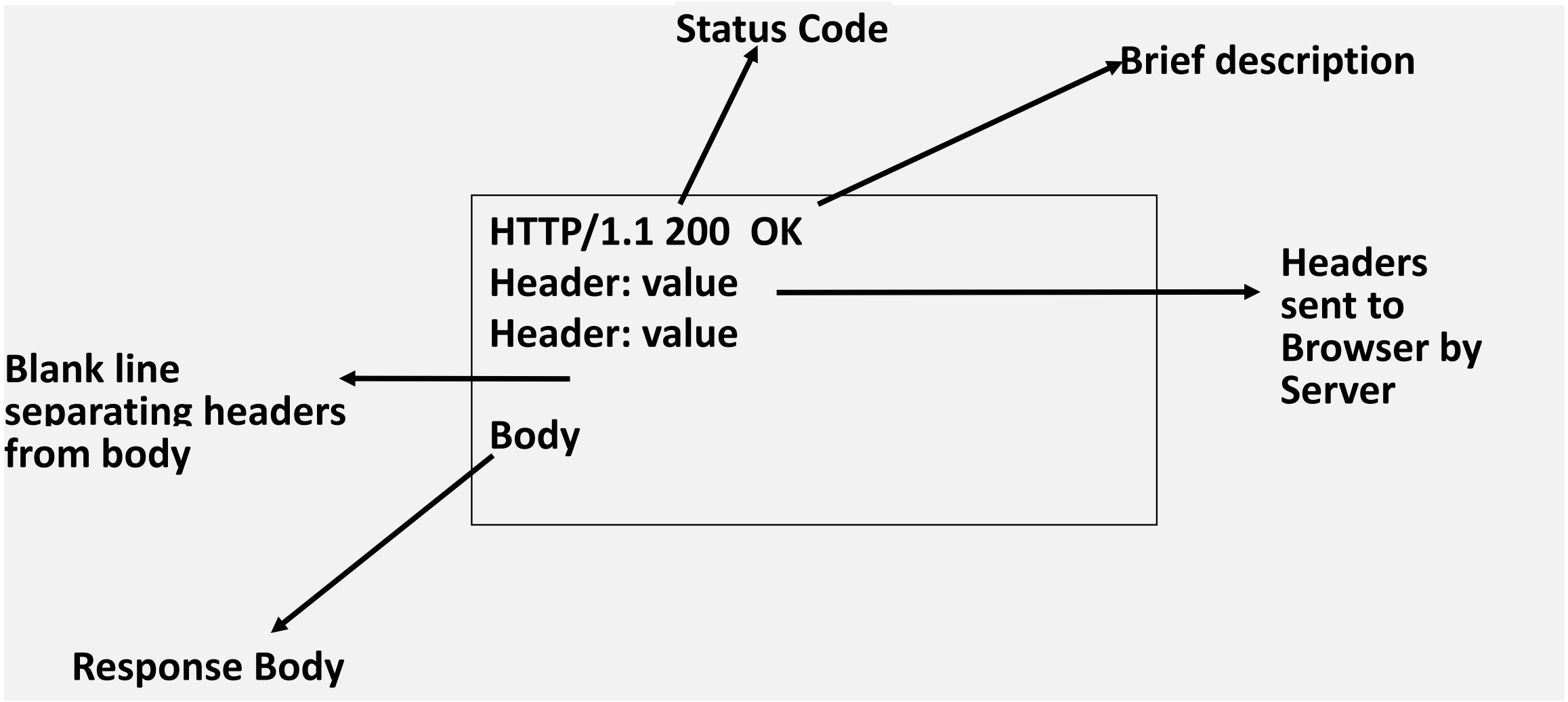
- We don’t have to let Django unpack the form’s fields; we can do it manually also.
- Each field is available as an attribute of the form using **{{form.name_of_field}}**, and in a Django template, will be rendered appropriately.
- Complete <label> elements can also be generated using the **label_tag()**.
- Errors related to a field can be displayed using **form.field.errors**.

```
Amount <br/>
{{form.amount}}
{# Display Errors related to Amount field #}
{% for error in form.amount.errors %}
   {{error}}
{% endfor %}
```

Http Request

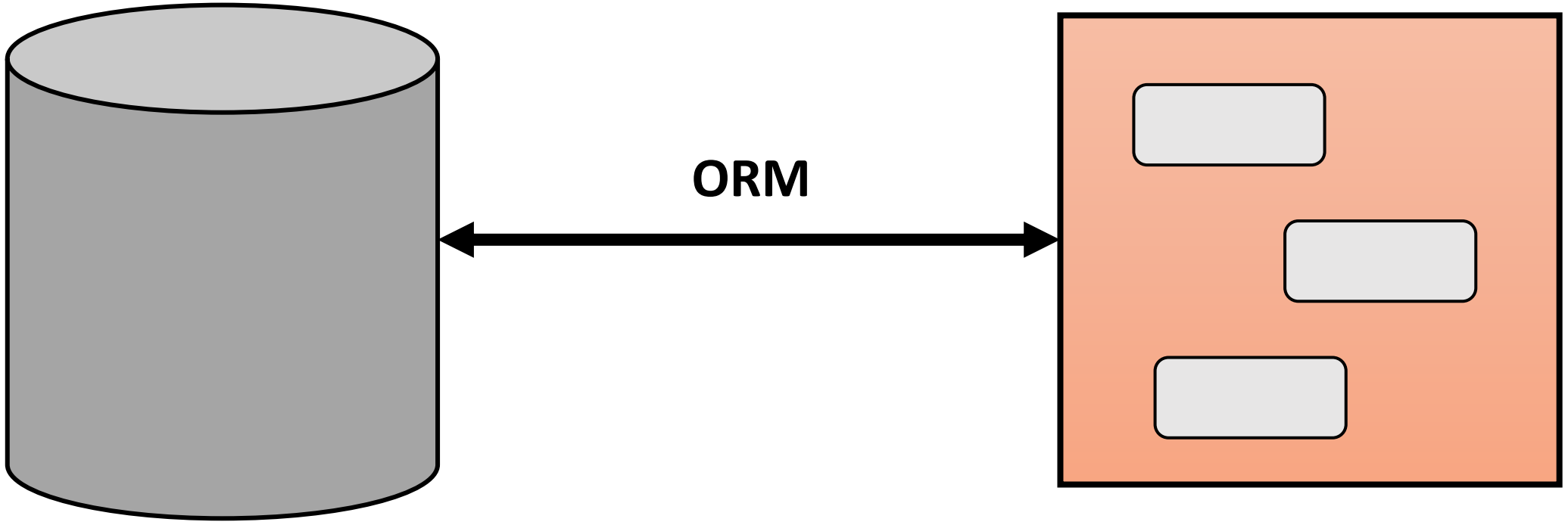


Http Response



Object Relational Mapping (ORM)

- ORM maps models (classes) to database tables
- Performs DML operations using simple methods without programmer having to write SQL commands
- Django ORM also creates database tables (known as migration) for models in applications



Steps related to ORM

- ❑ Create Model (class) along with fields
- ❑ Check whether models are valid
- ❑ Migrate models to Database
- ❑ Use ORM API to manipulate tables using models

Models

- A model is a class that contains data related to an entity.
- Each model is a Python class that subclasses **django.db.models.Model**.
- It contains the essential fields and behaviors of the data you're storing.
- Generally, each model maps to a single database table.
- Each attribute of the model represents a database column.
- Override **__str__** to return a string for model object.

Fields

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 (INTEGER, VARCHAR, TEXT).
- A field name cannot be a Python reserved word, because that would result in a Python syntax error.
- A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works.

Common Field Attributes

Attribute	Meaning
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.
default	The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.
primary_key	If True, this field is the primary key for the model.
unique	If True, this field must be unique throughout the table.

Book Model

books\Models.py

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=30,
                    unique=True, null=False)
    author = models.CharField(max_length=30, null=False)
    price = models.IntegerField(null=False)

    def __str__(self):
       return f"{self.id}-{self.title}-{self.author}-{self.price}"

    class Meta:
        db_table = 'books'
```

Validate Models

- Validate model classes in an application or entire project using CHECK command of manage.py.
- The following command checks whether models in books application are valid.

```
>python manage.py check books
System check identified no issues (0 silenced).
```

Create Migration Scripts

- Migration scripts are used to create or modify database objects.
- Command **makemigrations** of manage.py is used to generate migration scripts and place them in *migrations* folder of the application.

```
>python manage.py makemigrations books
Migrations for 'books':
  books\migrations\0001_initial.py
    - Create model Book
```

Migrate

- Command **migrate** of manage.py is used to execute migration scripts and apply changes to database.
- The following script creates BOOKS table in default database.

```
>python manage.py migrate books
Operations to perform:
  Apply all migrations: books
Running migrations:
  Applying books.0001_initial... OK
```

Database and Table

website\settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

```
CREATE TABLE books (
    id     INTEGER       NOT NULL
                         PRIMARY KEY AUTOINCREMENT,
    title  VARCHAR (30)  NOT NULL
                         UNIQUE,
    author VARCHAR (30)  NOT NULL,
    price  INTEGER       NOT NULL
);
```

Manager

- By default, Django adds a Manager with the name **objects** to every Django model class.
- A Manager is the interface through which database operations are provided to Django models.

Method	Description
all()	Returns a new QuerySet containing all objects in the table.
get()	Retrieves only one object based on the condition.
filter()	Returns QuerySet that contains objects that match the given lookup parameters.
exclude()	Returns QuerySet that contains objects that do NOT match the given lookup parameters.
create()	Creates and saves a new object in the database.

Adding an object

- Create an object of model class with required data
- Call **save()** to insert a row into corresponding table

```
>>>from books.models import Book
>>>b = Book(title="Outliers", author="Malcolm Gladwell", price=400)
>>>b.save()
```

```
>>> from books.models import Book
>>> Book.objects.create(title='The World Is Flat',
            author='Thomas Friedman',price=550)
<Book: 2 - The World Is Flat - Thomas Friedman - 550>
```

Update an object

- Retrieve object from database
- Make changes to fields of the object
- Call save() method

```
>>> book = Book.objects.get(id=2)
>>> book.price = 450
>>> book.save()
```

Deleting objects

- Retrieve object from database
- Call **delete**() method of object
- Returns number of rows deleted for each dependent object along with total objects deleted
- It is also possible to delete all objects that match the given field lookup using **delete**() method of QuerySet object

```
>>> book = Book.objects.get(id=3)
>>> book.delete()
(1, {'books.Book': 1})
```

```
>>> Book.objects.filter(author = 'Stephen King').delete()
(2, {'books.Book': 2})
```

Field Lookup

- Fields lookups are used to specify what objects are to be selected.
- They form the WHERE clause of SQL SELECT command

```
field__lookuptype=value
```

```
Book.objects.get(title='The Outliers')
Book.objects.filter(id__in=[1, 3, 4])
Book.objects.filter(id__range=(10,20))
Book.objects.filter(title__regex=r'^(T|S)')
```

Field Lookup

Lookup Type	Meaning
pk	Look up for primary key.
exact	An “exact” match.
iexact	A case-insensitive match.
contains	Case-sensitive containment test.
startswith, endswith	Starts-with and ends-with search, respectively.
in	In a given iterable; often a list, tuple, or queryset.
gt	Greater than.
gte	Greater than or equal to.
lt	Less than.
lte	Less than or equal to.
startswith	Case-sensitive starts-with.
istartswith	Case-insensitive starts-with.
endswith	Case-sensitive ends-with.
iendswith	Case-insensitive ends-with.

Field Lookup

Lookup Type	Meaning
range	Range test (inclusive).
date	For datetime fields, casts the value as date. Allows chaining additional field lookups. Takes a date value.
year	For date and datetime fields, an exact year match. Allows chaining additional field lookups. Takes an integer year.
month	For date and datetime fields, an exact month match. Allows chaining additional field lookups. Takes an integer 1 (January) through 12 (December).
day	For date and datetime fields, an exact day match. Allows chaining additional field lookups. Takes an integer day.
isnull	Takes either True or False, which correspond to SQL queries of IS NULL and IS NOT NULL, respectively.
regex	Case-sensitive regular expression match.

Ordering using order_by()

- It is possible to order objects in QuerySet using order_by() method

```
>>>Book.objects.all().order_by('price')
>>>Book.objects.all().order_by('author','-price')
>>>Book.objects.filter(price__gt=500).order_by('price')
```

Aggregation Functions

- Django provides aggregation functions in the **django.db.models** module.
- Functions must be imported before they are used.
- Functions are Avg, Count, Max, Min, Sum, StdDev, and Variance.
- Use **aggregate**() function with QuerySet object.

```
>>> from django.db.models import Avg, Max, Sum
>>> Book.objects.aggregate(Max('price'))
{'price__max': 750}
>>> Book.objects.filter(price__gte=400).count()
2
>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 425.0}
>>> Book.objects.all().aggregate(total=Sum('price'))
{'total': 1850}
```

Using static files

In order to use static files such as images, JavaScript and CSS in Django project, take the following steps:

- ❑ Make sure that **django.contrib.staticfiles** is included in your INSTALLED_APPS
- ❑ Store static files in a folder called **static** in your app. For example, /static/logo.jpg
- ❑ Use **{% load static %}** tag to load static resources
- ❑ Refers to static file using static tag. Ex : ****

Template Inheritance

- Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override.

books\templates\base.html

```
<html>
{% load static %}
<head>
  <link rel="stylesheet" href="{% static 'styles.css' %}" />
  <title>Books</title>
</head>
<body>
    <div class="banner">Books</div>
    {% block content%}{% endblock %}
    <hr/>
    <div class="footer">Srikanth Technologies </div>
</body>
```

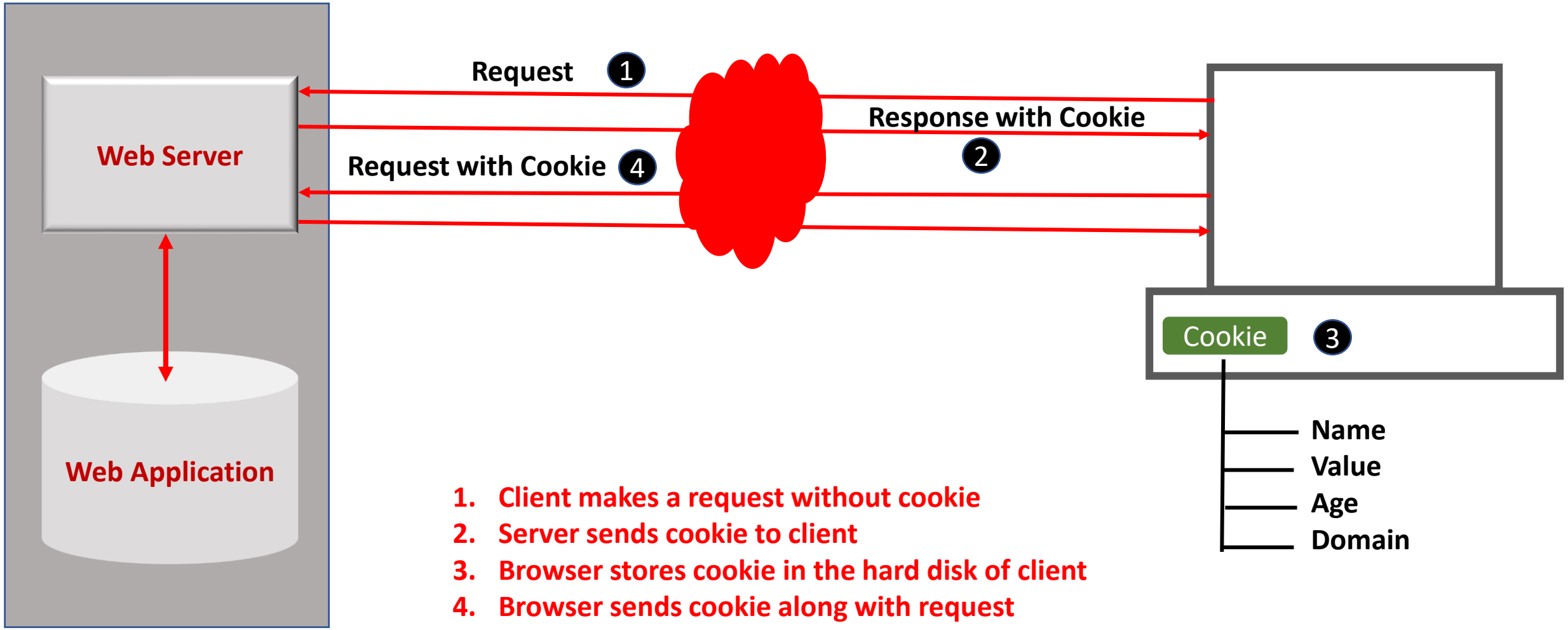
Template Inheritance

- Child template refers to base template using {% **extends** %} tag

books\templates\home.html

```
{% extends "base.html" %}
{% block content %}
<h2>Summary</h2>
.
.
.
{% endblock content %}
```

Cookies



1. Client makes a request without cookie
2. Server sends cookie to client
3. Browser stores cookie in the hard disk of client
4. Browser sends cookie along with request

Using Cookies

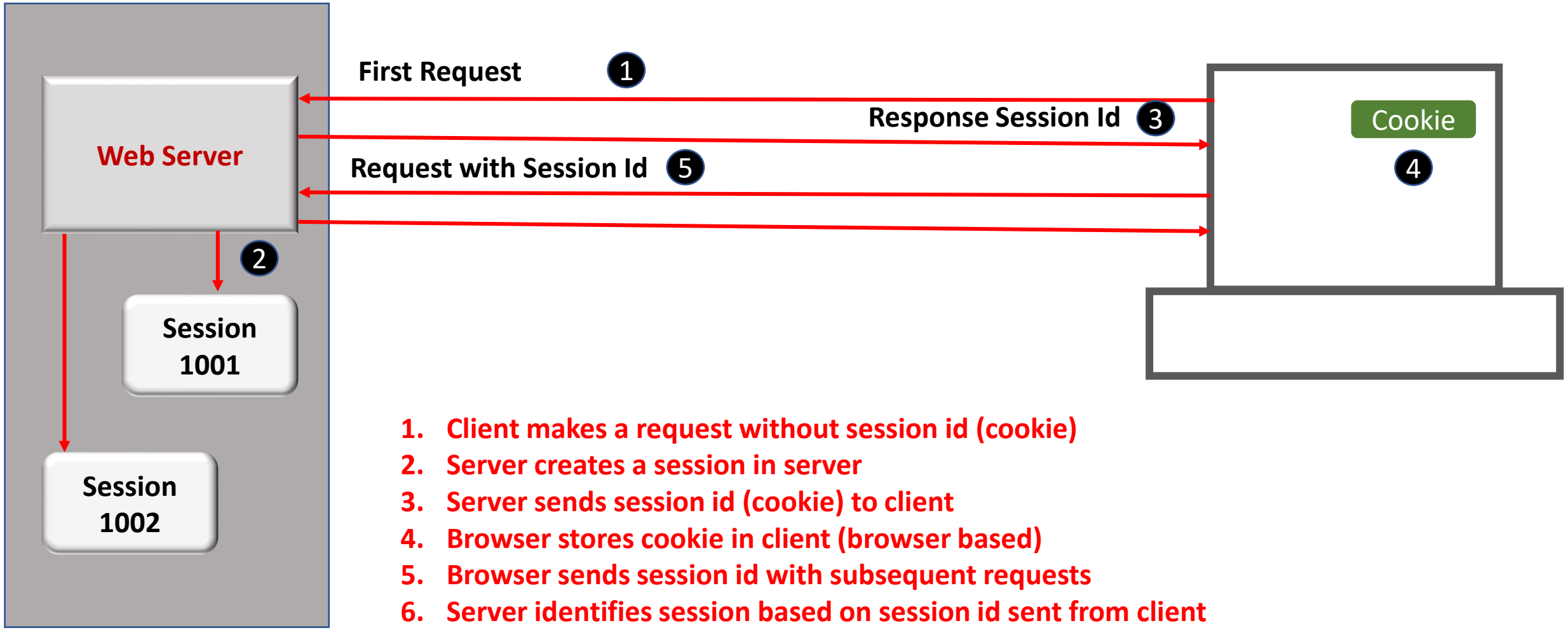
- Cookies are used to store data about client in client system.
- Programs in server send cookies to browser, which stores cookies either in hard disk or in browser's memory.
- Each cookie contains a name, value, domain, path and expiry date (except browser-based cookies).
- Django supports cookies using **COOKIES** attribute in **HttpRequest** object and **set_cookie**() method of **HttpResponse** object.

Creating and using Cookies

```
response.set_cookie(name, value,
  expires = datetime.datetime.now() + datetime.timedelta(days=10))
```

```
if 'city' in request.COOKIES:
   city = request.COOKIES['city']
else:
   city = ''
```

Sessions



1. Client makes a request without session id (cookie)
2. Server creates a session in server
3. Server sends session id (cookie) to client
4. Browser stores cookie in client (browser based)
5. Browser sends session id with subsequent requests
6. Server identifies session based on session id sent from client

Sessions

- Sessions allow applications to store data about clients in server.
- Each client gets a unique session on the server with an associated session id.
- Django provides access to session using **sessions** attribute of **HttpRequest** object.
- By default Django stores session data in a table in default database.
- We need to use migrate command with sessions application to create the table that is used to store sessions data, as shown below:

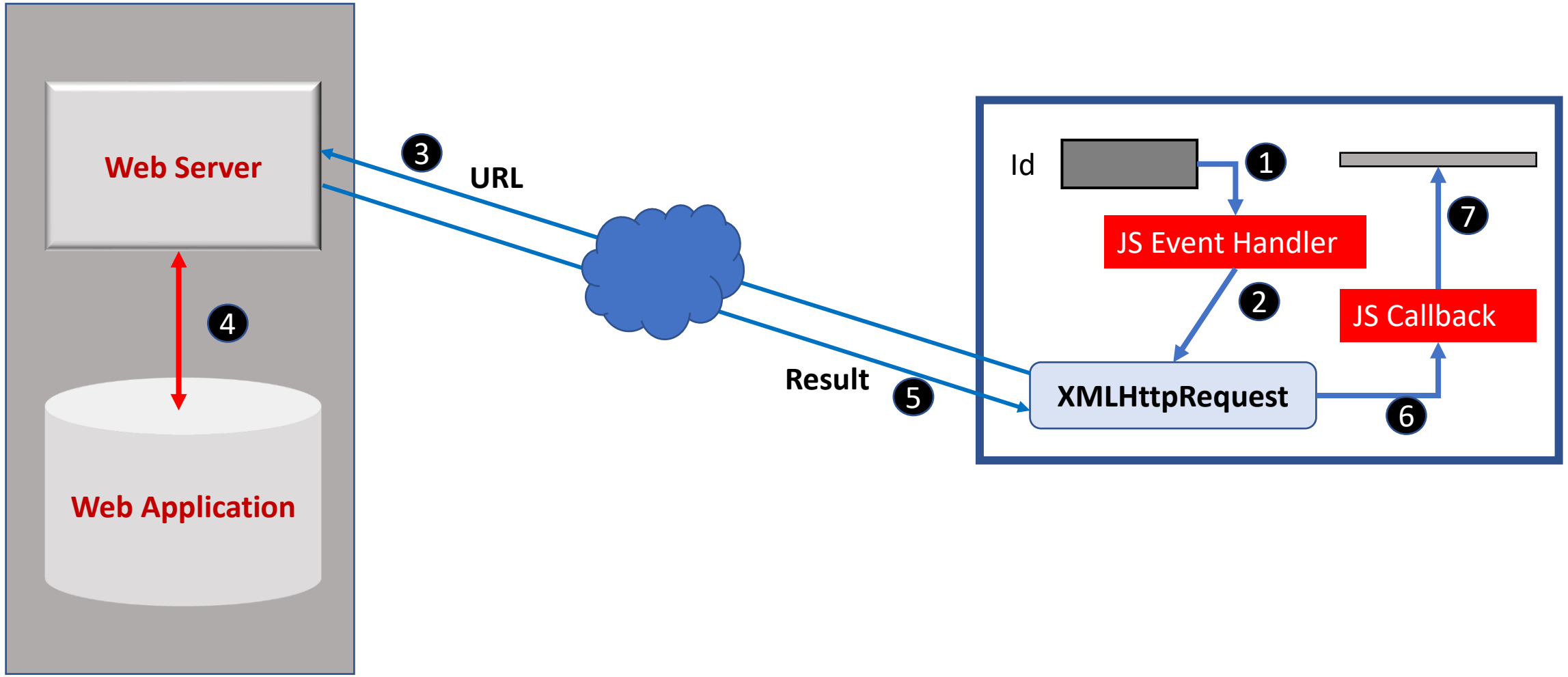
```
>python manage.py migrate sessions
```

Using Sessions

```
# Place a new key in session
request.session['names'] = names
```

```
# Check and use name if it is present in session
if 'names' in request.session:
     names = request.session['names']
else:
     names = []
```

AJAX (Asynchronous JavaScript And XML)



Advantages of AJAX

- Asynchronous
- Minimal data transfer
- Responsiveness
- Context is maintained
- No plug-in/code to be downloaded
- Several Toolkits and frameworks are available
- Tremendous industry acceptance

Making AJAX Call

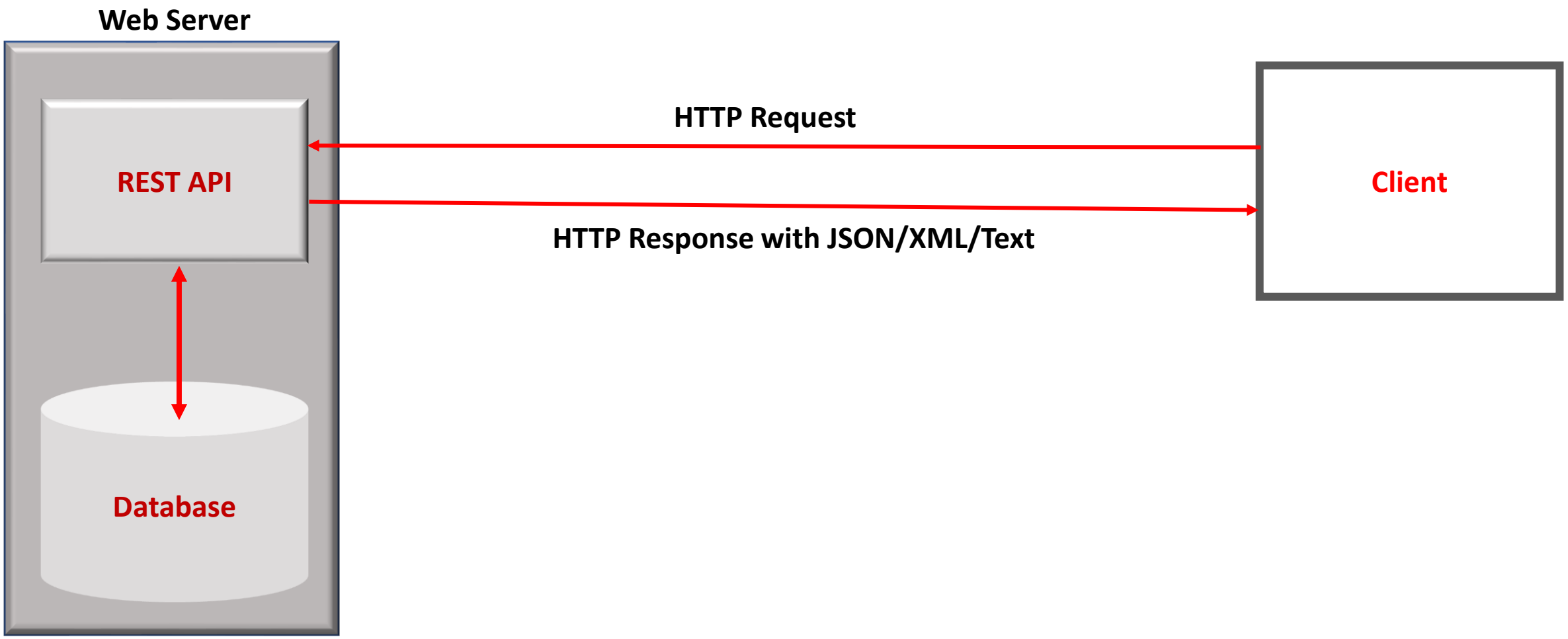
```
<script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js">
</script>
<script>
    function get_date_time()
    {
      // URL, params, callback
      $.get("/basics/datetime",null,show_date_time);  // Ajax Request
    }
    function show_date_time(response) {
      $("#datetime").html(response)
    }
</script>
<body>
 <h1>AJAX Demo</h1>
 <button onclick="get_date_time()">Show Date and Time</button>
 <h2 id="datetime"></h2>
</body>
```

Send Datetime from Server

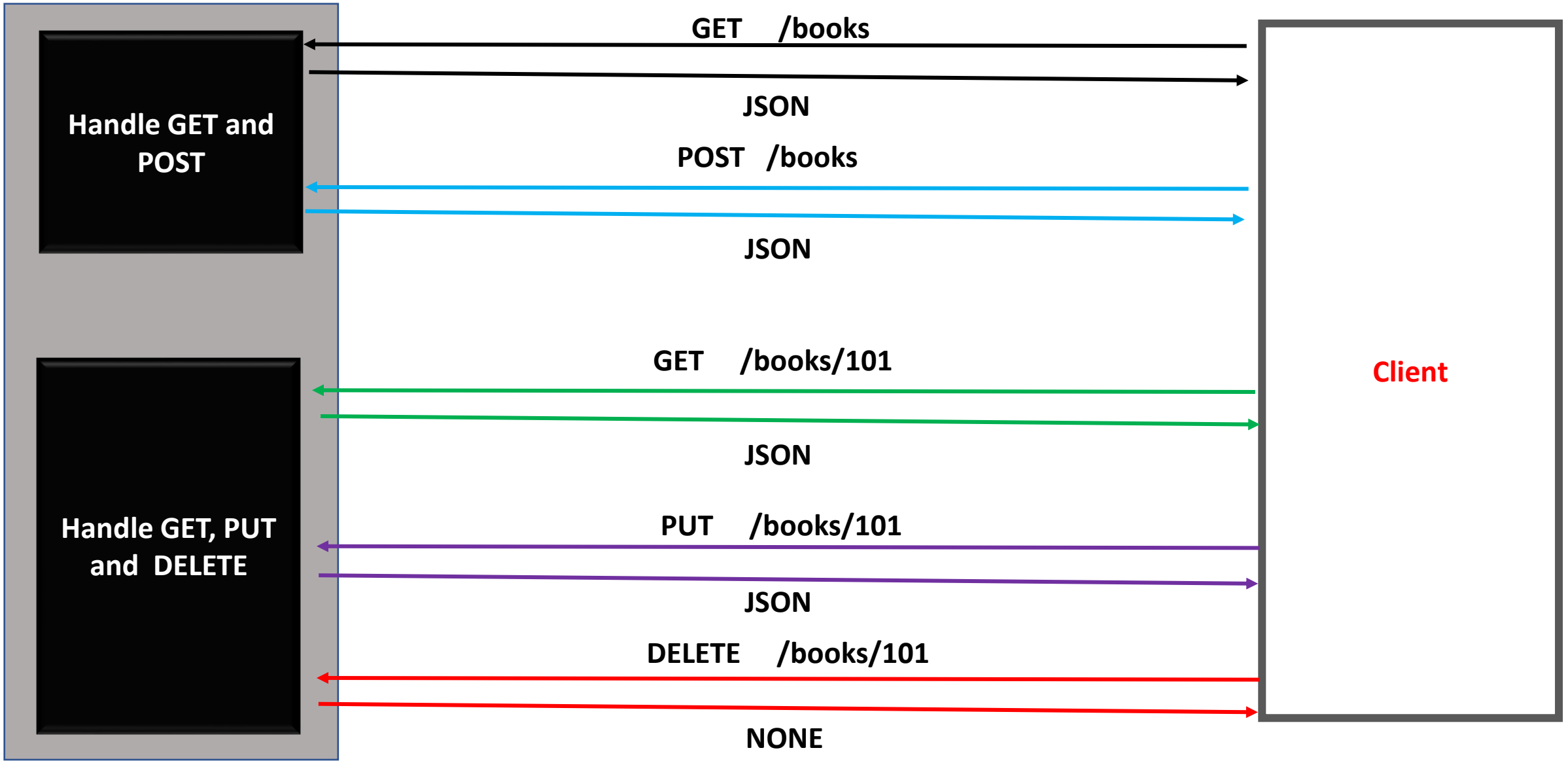
```
from django.http import HttpResponse
import datetime
import time

def send_datetime(request):
    # delay by 5 seconds
    time.sleep(5)
    return HttpResponse(str(datetime.datetime.now()))
```

REST API


Web Server
REST API
Database
HTTP Request
HTTP Response with JSON/XML/Text
Client

REST API Methods vs. Http Methods



Django Rest Framework

- Provides support for building Web APIs (RESTful Service).
- Provides serializer that converts Models to JSON back and forth.

```
>pip install djangorestframework
Collecting djangorestframework
  Downloading djangorestframework-3.12.1-py3-none-any.whl (913 kB)
     |                                                | 913 kB 2.2 MB/s
Requirement already satisfied: django>=2.2 in c:\python\lib\site-packages (from
djangorestframework) (3.1.2)
Requirement already satisfied: pytz in c:\python\lib\site-packages (from
django>=2.2->djangorestframework) (2019.3)
Requirement already satisfied: asgiref~=3.2.10 in c:\python\lib\site-packages
(from django>=2.2->djangorestframework) (3.2.10)
Requirement already satisfied: sqlparse>=0.2.2 in c:\python\lib\site-packages
(from django>=2.2->djangorestframework) (0.3.1)
Installing collected packages: djangorestframework
Successfully installed djangorestframework-3.12.1
```

Create Serializer

- A serializer is used to serialize and deserialize an object to and from JSON.
- Serializer is derived from **ModelSerializer** class and it specifies which fields of Model are to participate in serialization.

```
from rest_framework import serializers
from books.models import Book



# To perform JSON Serialization - Book to JSON
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ('id', 'title', 'author', 'price')
```

Processing GET and POST methods

```
from rest_framework import serializers
from rest_framework.decorators import api_view
from rest_framework.response import Response
from books.models import Book

@api_view(['GET', 'POST'])
def process_books(request):
    if request.method == "GET":
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)   # JSON Serialization
        return Response(serializer.data)
    else:  # Post
        book = BookSerializer(data=request.data) # JSON Deserialization
        if book.is_valid():
            book.save()  # insert into table
            return Response(book.data)
        else:
            return Response(book.errors, status=400)  # bad request
```

URLS related to Web API

```
# REST API views
path('rest/books/', rest_views.process_books),
path('rest/books/<int:id>', rest_views.process_one_book),
```

Making a GET request to WEB API

```
import requests

resp = requests.get("http://localhost:8000/books/rest/books")
if resp.status_code == 200:
   # Convert array of json objects to list of dict
   books = resp.json()
   for book in books:
      print(f"{book['id']} - {book['title']} - {book['author']} - {book['price']}")
else:
    print("Sorry! Could not get details of books!")
```

Making a PUT request to WEB API

```
import requests

id = input("Enter book id :")
price = input("Enter book price :")

res = requests.put(f"http://localhost:8000/books/rest/books/{id}",{'price': price})
if res.status_code == 200:
    print("Book was updated successfully!")
elif res.status_code == 404:
    print("Sorry! Book not found!")
else:
    print("Sorry! Could not update book due to error!")
```